Linear Temporal Logic - in Theory and Industry

by

Karin Greimel

A PhD Thesis
Presented to the Faculty of Computer Science in Partial Fulfillment of the Requirements for the PhD Degree

Assessors

Prof. Roderick Bloem (Graz University of Technology, Austria)
Prof. Werner Damm (Universität Oldenburg, Germany)

June 2016

Institute for Applied Information Processing and Communications (IAIK)
Faculty of Computer Science
Graz University of Technology, Austria

Abstract

Linear Temporal Logic specifications can be used for synthesis and formal verification of reactive systems.

For Linear Temporal Logic synthesis a Linear Temporal Logic specification is given as input and the output is a reactive system satisfying the specification. The specifications we consider are guarantees that need to be satisfied by the system if the inputs to the system satisfy a set of assumptions. In classic Linear Temporal Logic synthesis the system is allowed to behave arbitrarily if the assumptions are not satisfied. We argue that the system should be robust, it should try to satisfy the guarantees as well as possible even if the assumptions are not satisfied. We define two different notions of robustness, one for safety and one for liveness properties. In both cases a system is robust if a small number of violations of the assumptions lead to a small number of violations of the guarantees. For both, safety and liveness, we show how to verify and synthesize robust systems. Our work on synthesis of robust systems leads to new game theoretic results. For the synthesis of robust systems from safety specifications we define and solve ratio games. For the synthesis of robust systems from liveness specifications we develop a faster algorithm to solve Generalized Reactivity games.

For formal verification a Linear Temporal Logic specification and a reactive system are given as input and the output is a proof showing that the system satisfies the specification. We show how to use model checking, a formal verification technique, to verify specifications of smart cards. The proposed approach satisfies the Common Criteria Certification requirements for high Evaluation Assurance Levels. Common Criteria is a security certification scheme widely used in the smart card industry. Especially for e-government and banking products such as electronic passports, credit/debit cards, etc., high assurance of the security is of great importance. We give two case studies of security policy models using model checking. For the first case study we model a security IC. The model enabled the first Evaluation Assurance Level 6 certification within the German certification scheme. For the second case study we model the Java Card applet firewall. The model uncovered contradictions in the security requirements that have been overlooked for several years. We propose to integrate the formal verification method into the product design process by using UML statecharts as input to the model checker. UML is commonly known by engineers and thus allows a project to maximize the benefits of formal methods in the product design process.

Acknowledgements

I would like to thank my supervisor Roderick Bloem for introducing me to the field of formal methods, particularly for introducing me to the research community. I learned a lot about logic, automata, game theory and related topics but also how to conduct and present research. I am looking forward to learning a lot more and fruitful cooperation in the future.

I am grateful for all the fun and instructive times with my colleagues at the university. Special thanks go to Barbara Jobstmann, who has been a role model for me since I started working on my diploma thesis and who made me feel very welcome during my stay in Lausanne where we started working on synthesizing robust systems. I very much enjoyed working together. I would also like to thank Krishnendu Chatterjee for his contribution to the game theoretic part of synthesizing robust systems for liveness specifications. My thanks also go to all the colleagues at the institute, but mostly to Georg Hofferek and Robert Könighofer who are forthcoming and helpful and always open for discussions.

I would not have been able to finish my thesis without the support from people at NXP, thanks to Martin Schaffer, Christoph Herbst, Ernst Haselsteiner, Johannes Loinig, Hans-Gerd Albertsen and many others. I am also indebted to people outside NXP. Gerd Beuster initiated our work on security policy modeling and provided a first model. I got great technical support from Fraunhofer on COSIDE, especially from Norman Seßler. My thanks also go to Franz Röck for a first version of the Java Card OS firewall model. I hope for many more years of working together.

The work of my thesis was funded by Graz University of Technology, NXP Semiconductors, the European Commission with the projects COCONUT (FP7-2007-IST-1-217069) and DIAMOND (FP7-2009-IST-4-248613), and the Austrian Research Promotion Agency (FFG) with the project NewP@ss.

Last but not least I would like to thank my family for being there for me at all times; most important Wolfgang Aigner.

Big thanks to all who encouraged and supported me,

Karin Greimel
Graz, June 2016

Table of Contents

List of Tables

List of Figures

Glossary

applet An application running on a Java Card OS. 49, 50, 63, 68, 91

Bundesamt für Sicherheit in der Informationstechnik Common Criteria certification body. 47, 51

Common Criteria Security certification scheme. 2, 3, 45, 47, 48, 51, 52, 59, 60, 73, 78–80, 88–92, 94, 95

Evaluation Assurance Level Assurance level of a Common Criteria certification. 2, 3, 45, 47, 54, 61–63, 88–91, 95

flash A programmable non volatile memory. 49, 50
Functional Specification The functional specification of the product under evaluation. 53, 54, 59, 60, 62, 63, 68, 90

Java Card OS An operating system that allows to run applications written in Java on a smart card. v, 49–51, 63, 68

Protection Profile Template for a Security Target for a Common Criteria certification. 48, 51, 53, 54, 72, 74, 77, 78

Security Function Policy Set of Security Functional Requirements that define a policy. 74, 75, 77
Security Functional Requirement Security requirement stated in the Security Target for a Common Criteria certification. 52, 59–63, 65, 68, 73–75, 78, 89
Security Policy Model Formal model of security policies for Common Criteria certification. 2, 3, 45, 47, 48, 52–54, 59, 60, 63, 68, 88–90, 94, 95
Security Target Common Criteria document describing the Target of Evaluation and listing the Security Functional Requirements. 51–54, 59–61, 63, 65, 67, 89
smart card An Integrated Circuit with memory and applications, for example a credit card. xi, 2–4, 48, 49, 51, 56, 59, 68, 79–81, 92, 93
Special Function Register Register in an Integrated Circuit used to configure the circuit. 61
statechart A graphical representation of a finite state machine, part of UML. 2–4, 48, 56–59, 61, 63, 78, 79, 82, 88, 95

Target of Evaluation Product under evaluation. ix, 51–54, 62–64, 67, 89

Target of Evaluation Design Design description of the product under evaluation. 53, 54
Target of Evaluation Security Functionality Description of the security functionality of the product under evaluation. 51–53, 62, 74, 75, 77
Target of Evaluation Security Functionality Interface Interface description of the security functionality of the product under evaluation. 53, 63

waterfall model The waterfall model is a sequential design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of conception, initiation, analysis, design, construction, testing, production, and maintenance.. 53

Acronyms

AIS Application Notes and Interpretation of the Scheme. 63
API Application Programming Interface. 50

CASE Computer Aided Software Engineering. 89
CNF Conjunctive Normal Form. 31
CPU Central Processing Unit. 49, 50, 64
CTL Computation Tree Logic. 9, 10, 56, 57, 62, 63, 77, 82, 95

DCB objective Disjunctions of Conjunctions of Büchi objectives. 30–33
DNF Disjunctive Normal Form. 31

EEPROM Electrically Erasable Programmable Read Only Memory. 49, 50

FSM Finite State Machine. 56, 57, 59, 61, 63

GR(1) Generalized Reactivity(1). 1, 3, 7, 11, 13, 27, 30, 33, 35, 39, 42, 43
GUI Graphical User Interface. 94

IC Integrated Circuit. xi, 45, 47–51, 54, 59, 61, 63–65, 68, 88, 95
ICAO International Civil Aviation Organization. 50
ISM Interactive State Machine. 89
IT Information Technology. 45, 52

JCRE Java Card Runtime Environment. 50, 68, 69, 71, 73, 76, 90
JCVM Java Card Virtual Machine. 50, 69, 70, 89, 90

LTL Linear Temporal Logic. 1, 2, 9, 10, 37, 41, 43, 56, 57, 62, 63, 82, 95, 97, 98

MMU Memory Management Unit. 3, 49, 50, 61, 64, 65, 67
MSCC Maximal Strongly Connected Component. 32, 33

RAM Random Access Memory. 49, 50
ROM Read Only Memory. 49, 50
RTL Register Transfer Level. 1, 2

UML Unified Modeling Language. 2–4, 48, 56–58, 61, 63, 78, 79, 88, 90, 93–95, 97

1 Introduction

Linear Temporal Logic (LTL) is a formal language which was introduced by Pnueli in [120] for specifying and verifying the correctness of reactive systems. Reactive systems are continuously running systems that react to a given input. Examples are operating systems, controllers, and protocols. Correctness for such systems can not be defined in terms of the input-output relation at the end of the computation but are defined in terms of the input-output relation during computation. Temporal logic provides the means to define the correct input-output relation during computation; it allows to define sets of computation sequences.

A formal specification language can be used in the design of reactive systems in different ways. From an automation perspective LTL synthesis [47] is the most attractive application. The user provides a formal specification and the synthesis tool generates a system that satisfies the specification. Instead of writing Register Transfer Level (RTL) code the designer only needs to provide an LTL specification. Since the system is constructed from the specification it is correct by construction, assuming that the synthesis tool works correctly. Unfortunately LTL synthesis is a very complex problem, it is 2EXPTIME-complete [125]. The complexity issue can be circumvented by either considering small problems as in program repair [96] or by considering a subset of LTL such as Generalized Reactivity(1) (GR(1)) [119]. Apart from the complexity issue, at the current state of the art, the applicability of LTL synthesis for larger systems is also difficult because writing a good specification is not an easy task. The specification must be complete but should not over-constrain the system. Additionally we want the synthesized systems to not only satisfy the specification but to also have properties like minimum size, maximum speed, robustness and many more depending on the application field. Although LTL synthesis has been theoretically solved for many

years there is still a lot of research necessary before it can be applied in an industrial setting.

In contrast to LTL synthesis, LTL verification has found its way into industry in some application fields. The most common formal verification technique is model checking [50]. A model checker is a tool that takes a model and a specification as input and returns true if the model satisfies the specification and false otherwise. In this context a model is a finite state machine. A finite state machine can have different representations. Examples are hardware description languages such as Verilog and VHDL, proprietary model checking languages such as NuSMV and AIGER, or graphical representations such as Unified Modeling Language (UML) statecharts. A model checker gives a mathematical proof that the model satisfies the specification. In contrast to testing methods, which test a finite number of execution paths, a model checker shows that every execution path in the model satisfies the specification. In the semiconductor industry model checking is usually applied to RTL code on module level to verify parts of safety or security critical products. Formally verifying a large system at RTL level is usually too expensive since LTL model checking is in PSPACE [130].

The main advantage of having a formal specification independent of its usage is that it gives a clear and unambiguous documentation of the design intent. Specifications are often given in natural language, which is easier to understand than a formal language but also easier to misinterpret. A formal language leaves no room for interpretation, therefore it is a good addition to a natural language specification.

Formal methods in general aim at providing more rigorous methods in system design. They allow to find errors and contradictions in the specification early in the design process. With the increasing complexity of todays systems it becomes impossible to design a correct system without the help of such rigorous methods. Unfortunately formal methods are quite expensive in terms of expertise, time and computing power. Thus they are usually only applied to security or safety critical systems, where the costs of a failure of the system exceeds the costs of applying more rigorous design methods. Examples can be found in the automotive, aviation, space, and smart card industry. These industries often have to comply with standards or certification schemes which at some point require a formal specification and/or proof for security or safety critical parts of a product. An example of a security certification scheme which is widely accepted in the smart card industry is Common Criteria [36]. Common Criteria provides different Evaluation Assurance Levels. Higher Evaluation Assurance Levels require more formal evidence from the developer.

The contribution of this thesis is twofold. One part advances the field of LTL synthesis. The other part introduces new methods for Security Policy Models for high assurance level Common Criteria certifications. A short introduction to each is given below.

The first part of this thesis deals with the problem of *synthesizing robust systems*. There are many correct solutions for every LTL synthesis problem. Many different systems can realize a set of specifications, but the system we want

to synthesize should not only be correct but also be robust. We propose to define robustness for such systems and to synthesize robust systems that are resilient to errors. We consider GR(1) specifications, where the system has to satisfy a set of guarantees if the environment satisfies a set of assumptions. A system is robust if small deviations from the assumed environment behavior only leads to small deviations in the guaranteed system behavior. There exist different ways of measuring deviations from expected behavior. We propose two different measures for two different types of properties, namely safety and liveness generalized reactivity properties. Safety robustness specifications are based on counting each error of the environment and the system. A system is more robust if the ratio between environment errors and system errors is smaller. We define ratio games to solve the synthesis problem for safety robustness specifications. We also show how to solve the verification problem for safety robustness specifications. Liveness robustness specifications are based on counting the number of violated assumptions and guarantees. A system is more robust if less guarantees are violated. We develop a new game theoretic algorithm to solve the synthesis problem for liveness robustness specifications. We also show how to solve the verification problem for liveness robustness specifications.

The second part of this thesis deals with *security policy modeling for smart cards*. For high level Common Criteria certifications a Security Policy Model is required. A Security Policy Model is a formal model of the security policies of a security product such as a smart card. We show how such a Security Policy Model can be realized with a model checker and describe two case studies. For the first case study we model the access control policy of the Memory Management Unit (MMU) of the NXP P60 Secure Smart Card Controller which was used for the Evaluation Assurance Level 6 certification of the product. It was the first certification in the German scheme that reached Evaluation Assurance Level 6. For the second case study we model the access control policy of the Java Card firewall. The formal verification of the Java Card firewall requirements uncovers contradictions. The case study demonstrates that without formal analysis contradictions can stay undetected in a requirements specification for several years. We also show how the implementation of a Security Policy Model can be integrated into an industrial design process using UML statecharts as modeling language.

The first part is a contribution to basic research of possible future synthesis methods. It is based on logic, complexity theory, automata theory, and game theory. The second part is a contribution to current industry methods and processes. It is based on formal methods, certification standards, and industry requirements. This thesis gives deep insights into the theoretical background of formal methods but also shows how to apply formal methods in an industrial setting.

1.1 Outline

The thesis is presented in two main parts.

The first part given in Chapter 2 deals with the synthesis of robust systems. Starting with the problem statement in Section 2.1 and the description of the contribution in Section 2.2. Section 2.3 gives the necessary definitions for the subsequent sections. Section 2.4 presents a robustness definition for safety specifications and describes a solution to the corresponding verification and synthesis problem. Section 2.5 presents a robustness definition for liveness specifications and describes the solution of the corresponding verification and synthesis problem. Related work is given in Section 2.6 and the conclusions are given in Section 2.7.

The second part given in Chapter 3 deals with security policy models for smart cards. Starting with the problem statement in Section 3.1 and the description of the contribution in Section 3.2, Section 3.3 gives the necessary definitions for the subsequent sections. Section 3.4 describes the proposed security policy modeling approach and gives two case studies which describe security policy models for smart cards. Section 3.5 shows how security policy modeling can be integrated into the design process of a smart card by using UML statecharts. Related work is given in Section 3.6 and the conclusions are given in Section 3.7.

Chapter 4 gives an overall conclusion of the work.

2 Synthesizing Robust Systems

This chapter describes our work on synthesizing robust systems. It includes the work of two publications: [27] and [22].

2.1 Problem Statement

Current verification and synthesis approaches consider the functional correctness of a system as a Boolean question: either the specification is fulfilled, or it is not. This approach is unsatisfactory in many situations [23]. In particular, many specifications consist of environment assumptions and system guarantees. For such specifications, the classical approach does not impose any restrictions on the behavior of the system when the environment assumptions are *not* fulfilled. We argue that (1) desirable systems act in some "reasonable" way, even if the environment does not always fulfill the assumptions and (2) it is an undue burden on the user to specify the proper behavior of the system for each and every environment behavior. Desirable systems should fulfill a natural "graceful degradation" property in the sense that the system should fulfill the guarantees as well as it can, given any behavior of the environment.

Suppose that a system is required to accept up to 1000 requests per second and to respond to each request within 0.1 seconds. What should the system do when request number 1001 arrives? There are several options, including terminating the system, dropping the extra request, or delaying a response. Clearly, all of these approaches satisfy the specification, but some are better than others. (Cf. [56].) Thus, a system should not only be correct, it should also be *robust*, meaning that it "behaves 'reasonably', even in circumstances that were not anticipated in the requirements specification [...]" [79].

2.2 Contribution

We define robustness for safety properties and for liveness properties of the form $A \rightarrow G$, where A is an environment assumption and G is a system guarantee. For both safety and liveness properties we show how to verify and synthesize a robust system.

Let us first discuss the difference between safety and liveness properties. Here we only give an informal explanation and a small example, for a more detailed discussion refer to [4]. Safety properties stipulate that "bad things" do not happen. For example consider the controller of a traffic light. A possible safety property could be: "Exactly one light must always be on". In contrast, liveness properties stipulate that "good things" do happen eventually. Considering the controller of a traffic light again, a possible liveness property could be: "Eventually the green light must be on". The fundamental difference in verification and synthesis is that for safety properties it is always (at any point in time) possible to determine whether the property has been violated or not but for liveness properties this is not possible. Consider the traffic light again, as long as only one light is on at a time the above safety property is satisfied, but as soon as no light is on or more than one light is on the above safety property is not satisfied anymore. Now imagine the red light is on, currently we can not decide if the above liveness property is violated or not. Because of this intrinsic difference between safety and liveness properties we define two different notions of robustness.

Safety

For safety we define robustness as the ratio between the possible errors of the environment and the resulting errors of the system. For safety specifications we can define a measure to count how often/badly an assumption or a guarantee is violated. An environment error corresponds to a violation of an assumption and a system error corresponds to a violation of a guarantee. A system is robust if finitely many environment errors never lead to infinitely many system errors, it is k-robust if the ratio of environment errors to system errors is always smaller than or equal to k.

We give a solution to the verification problem and the synthesis problem for (k-)robust systems. The synthesis problem is solved through a novel type of games, we call ratio games. An optimal robust system corresponds to the winning strategy of a ratio game, where the system minimizes the ratio of system errors to environment errors. We show that ratio games have optimal positional strategies and show how to calculate an optimal positional strategy in pseudopolynomial time.

Liveness

For liveness the notion of robustness that we suggest aims to maximize the number of guarantees that are fulfilled for any number of assumptions that may

be violated. Note that for liveness properties we can not count how often a property is violated but only how many properties are violated.

We give different examples of robustness measures for Generalized Reactivity(1) (GR(1)) properties and show that they can all be reduced to Generalized Reactivity formulas. We show how to verify such formulas and how to synthesize them to robust systems. For synthesis we develop a novel game-theoretic algorithm that is faster than Zielonka's, although it does produce strategies with larger memory. Our algorithm can also be used for the synthesis of GR(1) properties, in which case it outperforms the algorithm of [119] when the state space of the specification is larger than the number of assumptions and guarantees.

2.3 Preliminaries

We discuss the verification and synthesis of reactive systems. The language of a reactive system consists of infinite words. For a word $w = w_1 \ldots$, let $|w| \in \mathbb{N} \cup \{\infty\}$ be the length of the word, let $w[..i] = w_1 \ldots w_i$ be the prefix of length i, and let $w[i..] = w_i w_{i+1} \ldots$ be the suffix starting at position i. We denote the set of all finite (infinite) words over the alphabet A by A^* (A^ω).

2.3.1 Systems

We consider systems with a set of input signals I and a set of output signals O. We define $AP = I \cup O$. We use the signals as atomic propositions in the specifications defined below. Our input alphabet is thus $\Sigma_I = 2^I$, the output alphabet is $\Sigma_O = 2^O$, and we define $\Sigma = 2^{AP}$.

Moore machines

We use Moore machines to represent systems. A *Moore machine* with input alphabet Σ_I and output alphabet Σ_O is a tuple $M = (Q, q_0, \delta, \lambda)$, where Q is the set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma_I \to Q$ is the transition function, and $\lambda : Q \to \Sigma_O$ is the output function. In each state, the Moore machine outputs a letter in Σ_O, then reads a letters in Σ_I, and moves to the next state. The *run* of M on a sequence $x = x_0 x_1 \ldots \in {\Sigma_I}^\omega$ is a sequence $\rho_0 \rho_1 \ldots \in Q^\omega$, where $\rho_0 = q_0$ and $\rho_{i+1} = \delta(\rho_i, x_i)$. The corresponding *word* is $\lambda(\rho) = w_0 w_1 \ldots \in \Sigma^\omega$, where $w_i = \lambda(\rho_i) \cup x_i$. The *language of* M, $L(M) \subseteq \Sigma^\omega$, consists of the words corresponding to the runs of M. We define $L^*(M) = L(M) \cap \Sigma^*$.

2.3.2 Acceptance Conditions

The specifications we use are automata and we synthesize a system that realizes a given specification using games. Both automata and games can have the following acceptance conditions.

Let Q be a set of states, an *acceptance condition* is a predicate $\mathrm{Acc} : Q^\omega \to \mathbb{B}$, mapping infinite runs to true or false (accepting and not accepting, or winning and losing, respectively). The *safety acceptance condition* is $\mathrm{Acc}(\rho) = \mathsf{true}$ iff

ρ never leaves $F \subseteq Q$, the set of safe states. The *Büchi acceptance condition* is $\mathrm{Acc}(\rho) = \mathsf{true}$ iff $\inf(\rho) \cap F \neq \emptyset$, where $F \subseteq Q$ is the set of accepting states and $\inf(\rho)$ is the set of elements that occur infinitely often in ρ. We abbreviate the Büchi condition as $\mathcal{B}(F)$. A *Generalized Reactivity acceptance condition* is a predicate $\bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{m_l} \mathcal{B}(A_{l,i}) \rightarrow \bigwedge_{i=1}^{n_l} \mathcal{B}(G_{l,i}))$, where $A_{l,i} \subseteq Q$ are assumptions and $G_{l,i} \subseteq Q$ are guarantees. To simplify notation, we will assume that the m_l are all equal to some constant m, and similarly for n_l and n. The acceptance condition is a *GR(1) acceptance condition* if $k = 1$, it is a *generalized Büchi acceptance condition* if $k = 1$ and $m = 0$, it is a *Streett acceptance condition* with k pairs if $m = n = 1$.

2.3.3 Automata

A deterministic *automaton* over the alphabet Σ is a tuple $A = (Q, q_0, \delta)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Sigma \rightarrow Q$ is the transition function.

A *run* of an automaton A on a word $w = w_0 w_1 \ldots \in \Sigma^* \cup \Sigma^\omega$ is the longest sequence $\rho(w) = \rho_0 \rho_1 \ldots \in Q^* \cup Q^\omega$ such that $\rho_0 = q_0$, and $\rho_{i+1} = \delta(\rho_i, w_i)$. An automaton accepts a word if its run is accepting (see below). We call the set $L(A)$ of infinite words accepted by A the *language of* A.

The *product automaton* $A = A_1 \times A_2$ of two automata is defined as usual.

Safety automaton

A *safety automaton* $A = (Q, q_0, \delta, F)$ is a complete deterministic automaton (Q, q_0, δ) together with a set $F \subseteq Q$ of *accepting states* such that there are no edges from non-accepting to accepting states. An infinite run is *accepting* if it never leaves F. See Section 2.3.2 for a formal definition of the safety acceptance condition.

Büchi automaton

A *Büchi automaton* $A = (Q, q_0, \delta, F)$ is an automaton with a Büchi condition with accepting state set F. An infinite run is *accepting* if a state in F is visited infinitely often. See Section 2.3.2 for a formal definition of the Büchi acceptance condition.

Single and Double Cost Automata

A *single (double) cost automaton* over the alphabet Σ is a tuple $C = (Q, q_0, \delta, c)$ consisting of a complete deterministic automaton (Q, q_0, δ) and a cost function $c : Q \times \Sigma \rightarrow \mathbb{N}$ ($c : Q \times \Sigma \rightarrow \mathbb{N} \times \mathbb{N}$, respectively) that associates to each transition a value in $\mathbb{N}$ ($\mathbb{N} \times \mathbb{N}$, resp.) called *cost*. In a double cost automaton, we use c_s and c_e to refer to the cost function of the first and the second component, respectively. The *maximal cost* is the smallest $W \in \mathbb{N}$ $\forall q \in Q$, $\sigma \in \Sigma : c(q, \sigma) \leq W$ ($c_e(q, \sigma), c_s(q, \sigma) \leq W$). The cost of a word $w \in \Sigma^* \cup A^\omega$, denoted by $C(w)$, is

the sum $\sum_{i=0}^{|W|} c(\rho(w)_i, w_i)$, For double cost automata, we use $C_e(w)$ and $C_s(w)$ to refer to the first and second component, respectively, of the cost of the word w.

The *sum of two cost automata* $A_1 = (Q_1, q_{01}, \delta_1, c_1)$ and $A_2 = (Q_2, q_{02}, \delta_2, c_2)$ is the cost automaton $A = A_1 + A_2 = (Q, q_0, \delta, c)$, where A is the product of the automata A_1 and A_2 with costs $c = c_1 + c_2$, i.e., $c((q_1, q_2), \sigma) = c_1(q_1, \sigma) + c_2(q_2, \sigma)$. The *product of two single cost automata* $A_1 = (Q_1, q_{01}, \delta_1, c_1)$ and $A_2 = (Q_2, q_{02}, \delta_2, c_2)$ is a double cost automaton $A = A_1 \times A_2 = (Q, q_0, \delta, c)$, where A is the product of the automata A_1 and A_2 with costs $c = (c_1, c_2)$, i.e., $c((q_1, q_2), \sigma) = (c_1(q_1, \sigma), c_2(q_2, \sigma))$.

2.3.4 Specifications

We use safety automata for safety specifications and Büchi automata for liveness specifications. In our examples, we also show Linear Temporal Logic (LTL) formulas describing the discussed properties. Furthermore we use LTL and Computation Tree Logic (CTL) specifications in Chapter 3 for model checking.

Temporal Logic

Temporal Logic is used to describe the behavior of reactive systems. LTL was introduced by Pnueli in [120] and CTL was introduced by Ben-Ari in [15] to describe properties of infinite computations where the specification of an end state is not possible. We will give a short introduction to the syntax and semantics of LTL future formulas. For further reading we recommend [106].

The *syntax of an LTL formula* over a set of atomic propositions AP is defined as follows.

1. Every atomic proposition p in AP is an LTL formula,
2. if f and g are LTL formulas then $\neg f$ and $f \wedge g$ are LTL formulas, and
3. if f and g are LTL formulas then $\mathsf{X}\, f$ and $f \,\mathsf{U}\, g$ are LTL formulas.

The operators used in the definition are denoted as the elementary operators. Other connectives can be seen as abbreviations: $f \vee g = \neg(\neg f \wedge \neg g)$, $f \rightarrow g = \neg f \vee g$, $f \leftrightarrow g = (f \wedge g) \vee (\neg f \wedge \neg g)$, $\mathsf{F}\, f = \mathsf{true}\,\mathsf{U}\, f$, and $\mathsf{G}\, f = \neg\,\mathsf{F}\,\neg f$.

The *semantics of an LTL formula* over a set of atomic propositions P are recursively defined over infinite words over the alphabet 2^{AP}.

1. Atomic Propositions:
   - $p \in AP : w[i..] \models p$ iff $p \in w_i$.
2. Boolean Operators:
   - $w[i..] \models \neg f$ iff not $w[i..] \models f$, and
   - $w[i..] \models f \wedge g$ iff $w[i..] \models f$ and $w[i..] \models g$.

3. Temporal Operators:
   - $w[i..] \models \mathsf{X}\, f$ iff $w[i+1..] \models f$, and
   - $w[i..] \models f\, \mathsf{U}\, g$ iff $\exists j \geq i : w[j..] \models g$ and $\forall k, i \leq k < j : w[k..] \models f$.

We refer to X as the next operator and to U as the until operator. The semantics of the operators always (G) and eventually (F) can be derived from the semantics of the elementary operators.

- $w[i..] \models \mathsf{G}\, f$ iff $\forall j \geq i : w[j..] \models f$, and
- $w[i..] \models \mathsf{F}\, f$ iff $\exists j \geq i : w[j..] \models f$.

A word w *satisfies* the formula f if $w \models f$. A Moore machine M satisfies or *realizes* an LTL formula f ($M \models f$) if all possible words of the Moore Machine satisfy the formula.

The *syntax of an CTL formula* over a set of atomic propositions AP is defined as follows.

1. Every atomic proposition p in AP is an CTL formula,
2. if f and g are CTL formulas then $\neg f$ and $f \wedge g$ are CTL formulas, and
3. if f and g are CTL formulas then $\mathsf{AX}\, f$, $\mathsf{A}(f\, \mathsf{U}\, g)$, and $\mathsf{E}(f\, \mathsf{U}\, g)$ are CTL formulas.

The operators used in the definition are denoted as the elementary operators. Other connectives can be seen as abbreviations: $f \vee g = \neg(\neg f \wedge \neg g)$, $f \rightarrow g = \neg f \vee g$, $\mathsf{EX}\, f = \neg\, \mathsf{AX}\, \neg f$, $\mathsf{EF}\, f = \mathsf{E}(\mathsf{true}\, \mathsf{U}\, f)$, and $\mathsf{AG}\, f = \neg\, \mathsf{EF}\, \neg f$.

The *semantics of an CTL formula* over a set of atomic propositions AP are recursively defined over the language of Moore Machines $M = (Q, q_0, \delta, \lambda)$ with the alphabet 2^{AP}. Let q be a state of M ($q \in Q$) and let $L(M, q)$ be the language of the Moore Machine with initial state q.

1. Atomic Propositions:
   - $p \in AP : (M, q) \models p$ iff $\exists w_0 w_1 \ldots \in L(M, q) : p \in w_0$.
2. Boolean Operators:
   - $(M, q) \models \neg f$ iff not $(M, q) \models f$, and
   - $(M, q) \models f \wedge g$ iff $(M, q) \models f$ and $(M, q) \models g$.
3. Temporal Operators:
   - $(M, q) \models \mathsf{AX}\, f$ iff $\forall$ runs $qq_1 \ldots$ of $M : (M, q_1) \models f$,
   - $(M, q) \models \mathsf{A}(f\, \mathsf{U}\, g)$ iff $\forall$ runs $qq_1 \ldots$ of $M : \exists j \geq i : (M, q_j) \models g$ and $\forall k, i \leq k < j : (M, q_k) \models f$, and
   - $(M, q) \models \mathsf{E}(f\, \mathsf{U}\, g)$ iff $\exists$ a run $qq_1 \ldots$ of $M : \exists j \geq i : (M, q_j) \models g$ and $\forall k, i \leq k < j : (M, q_k) \models f$.

We refer to E as the exists operator and to A as the for all operator. The semantics of the operators EX, EF and AG can be derived from the semantics of the elementary operators.

- $(M, q) \models \mathsf{EX}\, f$ iff $\exists$ a run $qq_1 \ldots$ of $M : (M, q_1) \models f$,
- $(M, q) \models \mathsf{EF}\, f$ iff $\exists$ a run $qq_1 \ldots$ of $M : \exists j : (M, q_j) \models f$,
- $(M, q) \models \mathsf{AG}\, f$ iff $\forall$ runs $qq_1 \ldots$ of $M : \forall j : (M, q_j) \models f$,

Safety

Safety properties are properties stating that something bad will never happen or equally that we always stay in a safe state. In temporal logic these properties are usually defined using the G operator. We use safety automata to specify safety properties. Given a safety automaton A, we say the Moore machine M *satisfies* A, if $L(M) \subseteq L(A)$. We use cost automata to specify the cost of an error for robust systems satisfying a safety specification.

Liveness

Liveness properties are properties stating that eventually some property will hold. We look at a special subset of properties, which can include safety and liveness. The specifications we consider are *GR(1) specifications*. GR(1) specifications consist of two parts: *assumptions* and *guarantees* [119]. They specify the interaction between an environment (controlling the input variables Σ_I) and a system (controlling the output variables Σ_O). The specification states that the system must fulfill all guarantees whenever the environment fulfills all assumptions.

A GR(1) specification over the alphabet Σ consists of m Büchi automata $A_1^a, \ldots, A_m^a$ for the environment assumptions and n Büchi automata $A_1^g, \ldots, A_n^g$ for the system guarantees [119]. Let $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$ be the product of all automata A_i^a and A_i^g, where the state space is $Q = Q_1^a \times \cdots \times Q_m^a \times Q_1^g \times \cdots \times Q_n^g$, the transition function is $\delta((q_1^a, \ldots, q_n^g), \sigma) = (\delta_1^a(q_1^a, \sigma), \ldots, \delta_n^g(q_n^g, \sigma))$, and the initial state is $q_0 = (q_{0,1}^a, \ldots, q_{0,n}^g)$. Let $J_i^a = \{(q_1^a, \ldots, q_n^g) \in Q \mid q_i^a \in F_i^a\}$ be the set of states that are accepting in A_i^a. Similarly, let J_i^g be the set of all states of $A^{GR(1)}$ that are accepting in A_i^g. The acceptance condition Acc is a GR(1) condition with assumptions J_i^a and guarantees J_i^g.

Note that the size of the state space of the specification grows exponentially with the number of assumptions and guarantees (if the Büchi automata have more than 2 states), whereas m and n grow linearly.

A system realizes a GR(1) specification $A^{GR(1)}$ if the language of the system is part of the language of $A^{GR(1)}$.

2.3.5 Games and Strategies

A *game graph* is a finite directed graph $G = (S, s_0, E)$ consisting of a set of states S, an initial state $s_0 \in S$, and a set of edges $E \subseteq S \times S$ such that each state has

at least one outgoing edge. The states are partitioned into a set S_1 of *Player-1 states* and a set S_2 of *Player-2 states*. When the initial state is not relevant, we omit it and write (S, E). A *play* $\rho = s_0 s_1 \ldots \in S^\omega$ is an infinite sequence of states such that for all $i \geq 0$ we have $(s_i, s_{i+1}) \in E$. We denote the set of all plays by Ω.

Given a game graph $G = (S, E)$, a *(finite memory) strategy* for Player 1 is a tuple (Γ, γ_0, π), where Γ is some (finite) set representing the memory, $\gamma_0 \in \Gamma$ is the initial memory content, and $\pi : S_1 \times \Gamma \to S \times \Gamma$ is a function mapping a Player-1 state s and a memory content to a successor state s' and an updated memory content such that $(s, s') \in E$. A Player-2 strategy is defined similarly. We denote by Π_1 and Π_2 the set of all possible Player-1 and Player-2 strategies, respectively. A strategy is *positional* if it depends only on the current state. We represent a positional strategy π for player p as a function from S_p to S. Let $\rho((\Gamma_1, \gamma_{0,1}, \pi_1), (\Gamma_2, \gamma_{0,2}, \pi_2), s)$ denote the unique play starting at s when Player 1 plays according to the strategy $(\Gamma_1, \gamma_{0,1}, \pi_1)$ and Player 2 plays according to $(\Gamma_2, \gamma_{0,2}, \pi_2)$.

A *game* is a game graph together with an objective. The game graph defines the possible actions of the players. The objective describes the goal for the players. Some games have a quantitative objective, for example mean payoff games. Other games have qualitative objective, namely that of winning. Generalized Reactivity games are games with quantitative objectives.

For quantitative objectives we define the value of a play, which is given by a *value function* $v : \Omega \to \mathbb{R} \cup \{-\infty, \infty\}$. The value of a state s under Player-1 strategy and Player-2 strategy, denoted by $v((\Gamma_1, \gamma_{0,1}, \pi_1), (\Gamma_2, \gamma_{0,2}, \pi_2), s)$, is the value of the play $\rho((\Gamma_1, \gamma_{0,1}, \pi_1), (\Gamma_2, \gamma_{0,2}, \pi_2), s)$.

We consider complementary objectives for the two players: Player 1 tries to minimize the value of a state and Player 2 tries to maximize it. (Note that the converse is more usual.) The Player-1 value of a state s under the strategy $(\Gamma_1, \gamma_{0,1}, \pi_1)$ is $\sup_{(\Gamma_2, \gamma_{0,2}, \pi_2) \in \Pi_2} (v((\Gamma_1, \gamma_{0,1}, \pi_1), (\Gamma_2, \gamma_{0,2}, \pi_2), s))$, which is the value Player 1 can guarantee with his strategy independent of the strategy Player 2 plays. A strategy $(\Gamma_1, \gamma_{0,1}, \pi_1)$ is optimal for Player 1 in state s if the Player-1 value of the state s under the strategy $(\Gamma_1, \gamma_{0,1}, \pi_1)$ is minimal. The Player-2 value and Player-2 optimal strategies are defined correspondingly. The value of a state s denoted by $v(s)$ is the Player-1 value of the play starting in s, in which both players play optimally.

A *mean payoff game* is described as a tuple $((S, s_0, E), w)$, where (S, s_0, E) is a game graph and $w : E \to \mathbb{N}$ is a *payoff function*. The value function for a play $\rho = s_0 s_1 \ldots$ in a mean payoff game is $v(\rho) = \limsup_{n \to \infty} \frac{1}{n} \sum_{i=0}^{n} w(e_i)$ with $e_i = (s_i, s_{i+1})$.

The objectives for games with qualitative objectives are acceptance conditions. Acceptance conditions are given in Section 2.3.2.

A *Generalized Reactivity (GR) game* is a tuple $((S, E), \mathrm{Acc})$, consisting of a game graph (S, E) and a Generalized Reactivity acceptance condition Acc. A play ρ is *winning* for Player 1 if it satisfies the objective of the game $\mathrm{Acc}(\rho) = \mathsf{true}$, otherwise it is winning for Player 2. A strategy π_1 is winning for Player 1 if for

all strategies π_2 of Player 2 the play $\rho((\Gamma_1, \gamma_{0,1}, \pi_1), (\Gamma_2, \gamma_{0,2}, \pi_2), s_0)$ is winning. A game is winning for Player 1 (Player 2) if there exists a winning strategy for Player 1 (Player 2, resp.). A *GR(1) game* is a game with a GR(1) acceptance condition. A *Streett game* is a game with a Streett acceptance condition.

Given a game graph G, two objectives are equivalent if all plays in G have the same winner for both objectives. The objectives are *equivalent* if they are equivalent for any game graph.

2.3.6 Synthesis

For synthesis we translate the specification into a game such that a winning strategy for the game corresponds to a Moore machine that satisfies the specification. In the following we describe how to translate automata into game graphs and GR(1) specifications into GR(1) games.

An automaton $A = (Q, q_0, \delta)$ over the alphabet Σ can be translated into a game graph (S, s_0, E) as follows. We define the set of Player-1 states as $S_1 = \{s_{(q,\sigma_i)} \mid q \in Q \text{ and } \sigma_i \in \Sigma_I\} \cup \{s_0\}$. The Player-2 states S_2 are given by the set $S_2 = \{s_{(q,\sigma_o)} \mid q \in Q \text{ and } \sigma_o \in \Sigma_O\}$. The set of game states is the set $S = S_1 \cup S_2$. Every state of the game (except for the initial state) represents a state of the automaton and an input or output label. Note that this corresponds to moving from a transition-labeled to a state-labeled system. Every outgoing transition of a state q in A is translated into two steps of the game: first, Player 1 chooses a letter σ_o from Σ_O by moving to the states $s_{(q,\sigma_o)}$, then Player 2 chooses a letter σ_i from Σ_I and moves according to the transition relation to a new state $s_{(q',\sigma_i)}$ such that $\delta(q, \sigma_o \cup \sigma_i) = q'$. Formally, we have that $E_1 = \{(s_{(q,\sigma_i)}, s_{(q,\sigma_o)}) \mid q \in Q, \sigma_o \in \Sigma_O, \text{ and } \sigma_I \in \Sigma_I\} \cup \{(s_0, s_{q_o,\sigma_o}) \mid \sigma_o \in \Sigma_O\}$, $E_2 = \{(s_{(q,\sigma_o)}, s_{(q',\sigma_I)}) \mid q, q' \in Q, \sigma_o \in \Sigma_O, \sigma_I \in \Sigma_I, \text{ and } \delta(q, \sigma_o \cup \sigma_i) = q'\}$, and $E = E_1 \cup E_2$.

A GR(1) specification $A^{GR(1)} = (Q, q_0, \delta, \text{Acc})$ over the alphabet Σ with m environment assumptions and n system guarantees can be translated into a GR(1) game $((S, s_0, E), ((A_1, \ldots, A_m), (G_1, \ldots, G_n)))$ as follows. The automaton (Q, q_0, δ) is translated into the game graph (S, s_0, E) as described above. Let J_i^a and J_j^g be defined as described for GR(1) specifications, then the assumption sets are $A_i = \{s_{(q,\sigma_o)} \mid q \in J_i^a\}$ and the guarantee sets are $G_j = \{s_{(q,\sigma_o)} \mid q \in J_j^g\}$.

2.4 Safety

This section proposes a formal notion of robustness through graceful degradation for discrete functional safety properties: A small error by the environment should induce only a small error by the system, where the error is defined quantitatively as part of the specification, for instance, as the number of failures. Given such a specification, we define a system to be robust if a finite environment error induces only a finite system error. As a more fine-grained measure of robustness, we define the notion of k-robustness, meaning that on average, the number of system failures is at most k times larger than the number of environment failures. We

show that the synthesis question for robust systems can be solved in polynomial time as a one-pair Streett game and that the synthesis question for k-robust systems can be solved using *ratio games*. Ratio games are a novel type of graph games in which edges are labeled with a cost for each player, and the aim is to minimize the ratio of the sum of these costs. We show that ratio games are positional, that the associated decision problem is in NP $\cap$ co-NP, and that they can be solved in pseudopolynomial time. They can be solved in polynomial time if the cost of a failure is assumed to be constant.

In the next subsection, we present our framework based on error functions, and define robustness and k-robustness. In Section 2.4.2, we introduce ratio games and show how to solve them. Section 2.4.3 shows how to use ratio games to construct correct and robust systems.

2.4.1 Defining Robustness

In this section we introduce our notion of robustness based on error specifications. We show how error specifications relate to classical specifications and the notion of realizability. We conclude with an example.

Definition 1. An *error function* is a function $d : \Sigma^* \cup \Sigma^\omega \to \mathbb{N} \cup \{\infty\}$. The function is monotonically increasing in the sense that if w' is a prefix of w then $d(w') \leq d(w)$.

The error functions define a distance between allowed and observed behavior, for instance, by measuring the number of failures in some appropriate sense. Thus, $d(w) = 0$ indicates that w fulfills the specification, and a higher value indicates a more serious violation of the specification.

Definition 2. An *error specification* is a pair of error functions (d_e, d_s).

Error specifications provide a measure of "badness" for both the environment behavior (using d_e) and the system behavior (using d_s) and form the specifications we use in the sequel. We assume that these specifications are provided by the user.

Definition 3. A Moore machine M *realizes* an error specification (d_e, d_s) if $\forall w \in L(M) : d_e(w) = 0$ implies $d_s(w) = 0$.

Thus, an error specification induces a classical specification $A \to G$, where $A = \{w \in \Sigma^\omega \mid d_e(w) = 0\}$ and $G = \{w \in \Sigma^\omega \mid d_s(w) = 0\}$ are sets of infinite words.

The following notion is an alternative to realizability, forbidding the system to make mistakes before the environment does.

Definition 4. A Moore machine M *strictly realizes* an error specification (d_e, d_s) if $\forall w \in L^*(M) : d_e(w[..|w| - 1]) = 0$ implies $d_s(w) = 0$. An error specification is *strictly realizable* if there exists a Moore machine that strictly realizes it.

Example 1. An example of a specification that is realizable but not strictly realizable is $A_1 \wedge A_2 \rightarrow G_1 \wedge G_2$, where x is an input, y is an output, A_1 requires that x is always true ($\mathsf{G}\, x$), A_2 says that x is initially equal to y ($x \leftrightarrow y$), G_1 states that y is always true ($\mathsf{G}\, y$), and G_2 states that x in the first step and y in the second step are different ($x \not\leftrightarrow (\mathsf{X}\, y)$). All Moore machines that realize the specification start with setting y to false, which violates the guarantees but forces the environment to do the same[1].

Definition 5. A Moore machine M is *robust* with respect to an error specification (d_e, d_s) if $\forall w \in L(M) : d_e(w) \neq \infty$ implies $d_s(w) \neq \infty$.

This means that a robust system can recover from a finite environment error. Note that a system can be robust with respect to a specification that it does not realize if it contains a word with a finite system error but no environment error. Error specifications can forbid words by assigning infinite system costs. (In particular, this is possible when such specifications are given by double cost automata, as below.)

In order to calculate the quality of a robust system we want to calculate the largest system error for every environment error.

Definition 6. A Moore machine M is *k-robust* with respect to an error specification (d_e, d_s) if $\exists d \in \mathbb{N} : \forall w \in L^*(M) : d_s(w) \leq k \cdot d_e(w) + d$.

Obviously, every k-robust system is robust, regardless of k. Also, every robust system is k-robust for some finite k, see Theorem 6, i.e., for every finite Moore machine, the growth of the system error is either linear with respect to the environment error or unbounded. This motivates our choice of the robustness measure as a linear function. The definition of k-robustness allows us to rank Moore machines with respect to error specifications: A smaller k is better, it means that the system error increases slowly with the environment error. The constant d allows the system finitely many system failures independent of the environment error. In this work, we focus on the infinite behavior of a machine, and note that d can be bounded by the product of the size of the Moore machine and the maximal weight. We leave minimization of d to future work.

Definition 7. A Moore machine *(k-)robustly (and strictly) realizes* an error specification if it (strictly) realizes the specification and it is (k-)robust with respect to the specification.

In the remainder, we use double cost automata to define error specifications. The environment (system) error function associated with C maps each $w \in \Sigma^* \cup \Sigma^\omega$ to its cost $C_e(w)$ ($C_s(w)$, respectively). Note that a double cost automaton can be seen as the product of two single cost automata. We can construct an error specification from a set of cost automata C_{A_i} for the system and C_{G_i} for the environment. The error specification (a double cost automaton) is the product of the sum of all C_{A_i} and the sum of all C_{G_i}.

[1] This specification is based on an example by Marco Roveri.

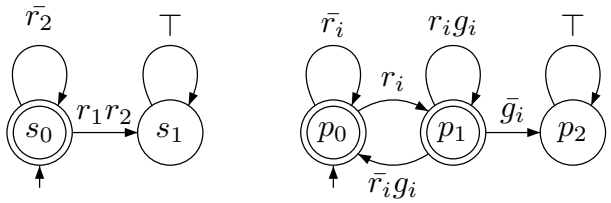


Figure 2.1: Automata for $A = \mathsf{G}(\neg(r_1 \wedge r_2))$ and $G_i = \mathsf{G}(r_i \rightarrow \mathsf{X}\, g_i)$

Example 2. Consider a system with two request signals r_1 and r_2 as inputs and two grant signals g_1 and g_2 as outputs. We want the system to respond to each request with a grant in the next step. Formally, we require that the system satisfies $G_i = \mathsf{G}(r_i \rightarrow \mathsf{X}\, g_i)$ for $i \in \{1, 2\}$. The system should also guarantee that grants are mutually exclusive, i.e., $G_3 = \mathsf{G}\,\neg(g_1 \wedge g_2)$. To avoid a contradicting specification, we assume that requests are also mutually exclusive, i.e., $A = \mathsf{G}\,\neg(r_1 \wedge r_2)$. Figure 2.1 shows two safety automata, one for A and one for G_1 and G_2. Note that we summarize labels on edges with Boolean expressions over r_i and g_i, where a horizontal alignment of two variables represents a conjunction and a vertical alignment of two variables represent a disjunction. We use a bar to denote negation and $\top$ to denote true. States depicted with two cycles are accepting states. Note that the automaton for G_3 is exactly the same as for A, where r_1 and r_2 are renamed to g_1 and g_2, respectively.

Starting from the specification $A \rightarrow (G_1 \wedge G_2 \wedge G_3)$, we can define what it means for the system and the environment to fail. In particular, the environment violates assumption A if it raises r_1 and r_2 at the same time. This corresponds to taking the edge from s_0 to s_1. The leftmost automaton C_A in Figure 2.2 is a cost automaton that counts every violation of the environment. Note that once the environment "pays" for taking the edge $r_1 r_2$, we go back to the initial state, resetting the specification. Similarly, if the system violates Guarantee G_i by choosing to go from p_1 to p_2, it also incurs cost 1 as shown in the second automaton C_{G_i} in Figure 2.2.

Note that it is up to the user to define the cost of a violation and the state in which to continue after the specification is violated. A reset or a skip are two natural alternatives. A reset corresponds to an edge to the initial state. For a skip, we simply add a self-loop. The rightmost automaton C'_{G_i} of Figure 2.2 is an alternative cost automaton for G_i with $i \in \{1, 2\}$, which uses a mixture of reset and skip. For the cost automaton C_{G_i}, the word $(r_1, \bar{g_1})(r_1, \bar{g_1})(\bar{r_1}, \bar{g_1})^\omega$ has cost 1 whereas it has cost 2 for the cost automaton C'_{G_i}. For the second automaton, the cost corresponds to the number of unanswered requests.

The costs on the edges are given by the user. For instance, the user might consider a violation of the mutual-exclusion properties G_3 more severe and associate with it a higher cost than a violation of the response properties G_1 or G_2.

Given cost automata C_{G_1}, C_{G_2}, and C_{G_3} that describe the cost and the

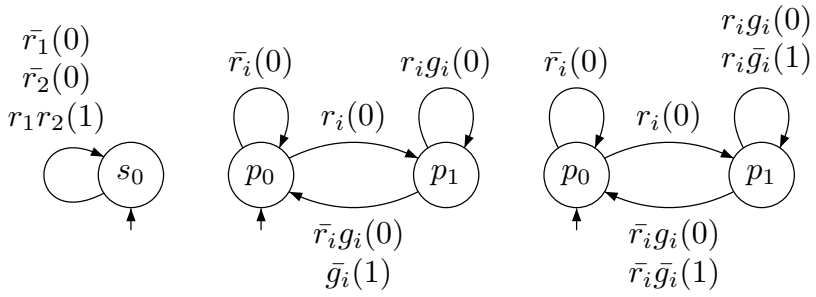


Figure 2.2: Cost automata C_A, C_{G_i}, and C'_{G_i} counting violations of A and G_i

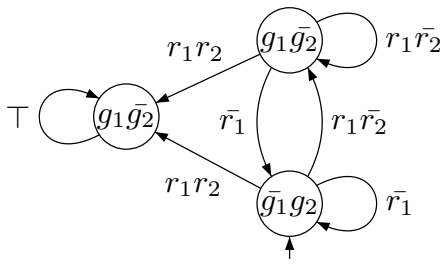


Figure 2.3: Automatically constructed system for $A \to (G_1 \wedge G_2 \wedge G_3)$

behavior associated with a violation of the corresponding property, we can construct a cost automaton $C_G = C_{G_1} + C_{G_2} + C_{G_3}$ for $G = G_1 \wedge G_2 \wedge G_3$. The automaton C_G defines the error function of the system. The cost automaton for the environment C_A specifies the error function of the environment. The product $C = C_A \times C_G$ is the error specification.

Figure 2.3 shows a system M (synthesized with Lily [94]) for the specification $A \to G$. It is easy to see that M satisfies $A \to G$. As long as the environment satisfies A, which means that it does not provide r_1 and r_2 simultaneously, the system responds to each r_i with the corresponding g_i in the next step. However, M is not robust with respect to C: The input sequence $i = (r_1r_2)(\bar{r_1}r_2)^\omega$ has cost one, but the corresponding output $o = (\bar{g_1}g_2)(g_1\bar{g_2})^\omega$ of the system has cost ∞.

Figure 2.4 shows two systems that are robust with respect to the error specification, for any word with finitely many environment errors the systems produce finitely many system errors. The first system in Figure 2.4 is 2-robust with respect to the error specification using C'_{G_i} whereas the second system in Figure 2.4 is 1-robust. For the input $(r_1r_2)^\omega$ the output of the first Moore machine is $(\bar{g_1}g_2)(\bar{g_1}\bar{g_2})^\omega$ and for the second it is $(g_1\bar{g_2})^\omega$.

Note that out of the three systems given above (which all satisfy $A \to G$) the

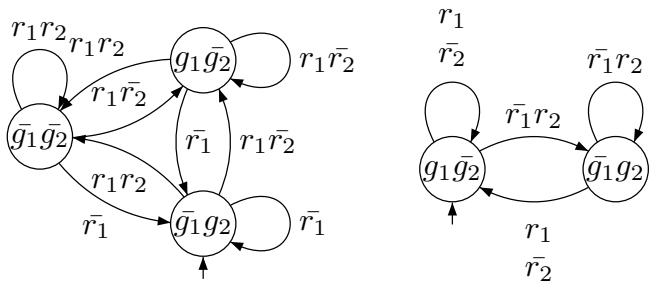


Figure 2.4: A 2-robust and a 1-robust system

last system is the most robust one. In our opinion, it is also the one most likely to please the designer.

In Section 2.4.3 we show how to synthesize (strictly) realizing robust and k-robust systems from an error specification. We also show how these notions can be verified. The next section introduces Ratio games, which are crucial to our synthesis algorithms.

2.4.2 Ratio Games

In this section we introduce ratio games, which we need to synthesize k-robust systems. Intuitively, a system is k-robust if the ratio of the system error to the environment error is smaller than or equal to k for every word of the system. An optimal strategy for Player 1 in a ratio game minimizes this ratio.

Definition 8. A *ratio game*[2] G is a tuple $((S, s_0, E), w_1, w_2)$ consisting of a game graph (S, s_0, E) and two weight functions $w_1, w_2 : E \to \mathbb{N}$ mapping edges to non-negative integer values. The value function for a play $\rho = s_0 s_1 \ldots \in S^\omega$ is

$$v(\rho) = \lim_{m\to\infty} \limsup_{l\to\infty} \frac{\sum_{i=m}^{l} w_1(s_i, s_{i+1})}{1 + \sum_{i=m}^{l} w_2(s_i, s_{i+1})} \tag{2.1}$$

Ratio games are a generalization of mean payoff games. If $w_2(e) = 1$ for all $e \in E$, then G is a mean payoff game. Note that the sequence of quotients for $l \to \infty$ might diverge, which requires the use of lim sup or lim inf. We follow the definition of mean payoff games and take the lim sup. The outer-most limit ensures that only the infinite behavior is relevant as in the definition of k-robustness, i.e., if $\sum_{i=0}^{\infty} w_1(e_i)$ is finite, then $v(\rho) = 0$. The addition of 1 in the denominator avoids division by zero. It does not influence the value of $v(\rho)$ if $\sum_{i=0}^{\infty} w_2(e_i)$ is infinite.

[2]Our graph-based ratio games should not be confused with those of [129], which represent games in a normal form, enumerating all strategies. We cannot use that representation to obtain a polynomial algorithm.

The *maximal weight* W in a ratio game $((S, s_0, E), w_1, w_2)$ is defined by $W = \max\{w_i(e) \mid e \in E, i \in \{1, 2\}\}$. The set of possible values $v(\rho)$ of a play ρ, where both players play positional strategies is $V = \{0, \frac{1}{|S|\cdot W}, \ldots, \frac{|S|\cdot W}{1}, \infty\}$. Lemma 1 shows that ratio games have optimal positional strategies, which implies that it suffices to consider positional strategies and that the value of every state is in V.

Lemma 1. *Ratio games have optimal positional strategies.*

Proof. It suffices to show that the two one-player games ($S_2 = \emptyset$, respectively $S_1 = \emptyset$) have optimal positional strategies [80]. Consider a game graph G with $S_2 = \emptyset$. Take in G a simple cycle with the minimum ratio r of all simple cycles. We show that the positional strategy π_1 that goes to this simple cycle and stays within it forever is optimal. Note that the value $v(\rho)$ of the play ρ induced by the strategy π_1 is r, since the outer-most limit in Eq. 2.1 allows us to ignore a finite prefix of ρ.

If $r = 0$, the claim trivially holds. If $r = \infty$, then in any simple cycle the sum of the weights w_2 is 0 and the sum of the weights w_1 is strictly greater than 0. This implies that all edges e on cycles have weight $w_2(e) = 0$ and in every cycle there is at least one edge e with $w_1(e) > 0$, and so any infinite play has ratio ∞.

For $0 < r < \infty$, let r be $\frac{a}{b}$ for some integers $a, b > 0$ and let ρ' be an arbitrary play in the single player game. We decompose ρ' into a sequence of ratios $\frac{a_1}{b_1}, \frac{a_2}{b_2} \ldots$ by the following procedure (cf. [140]): we put the states of ρ' on a stack in the order of their appearance, once we encounter a state q that is already on the stack, we remove the sequence from the first to the second appearance of q and compute its ratio $\frac{a_i}{b_i}$. Note that the sum of the weights w_1 and w_2 in this cycle can be c_i-times larger than a_i and b_i, respectively, where c_i is some integer constant between 1 and $W \cdot |S|$. Note that the height of the stack is at most $|S|$. Due to the outer-most limit, we can ignore the part of ρ' that is always left on the stack in the computation of the value $v(\rho')$. Then, $v(\rho') = \limsup_{l\to\infty} \frac{\sum_{i=1}^{l} c_i \cdot a_i}{1+\sum_{i=1}^{l} c_i \cdot b_i}$ for some constants $0 < c_i \leq W \cdot |S|$. Since the minimum simple-cycle ratio is $\frac{a}{b}$, we know that $\frac{a_i}{b_i} \geq \frac{a}{b}$ for all $i > 0$ and together with the fact that c_i's are positive integer constants, we know that $v(\rho') \geq \limsup_{l\to\infty} \frac{\sum_{i=1}^{l} a}{1+\sum_{i=1}^{l} b}$ and therefore $v(\rho') \geq \frac{a}{b}$.

The proof for Player-2 games is analogous. □

The decision problem of a ratio (mean payoff) game is, given a ratio r (mean payoff v) decide if the value of the game is at least r (v). The decision problem for mean payoff games is in NP ∩ co-NP [140]. We show how the decision problem for ratio games can be reduced to the decision problem of mean payoff games. The reduction shows that the decision problem for ratio games is in NP ∩ co-NP. We also use this reduction to calculate the values of the states in a ratio game. The reduction is similar to that used by Lawler [102] for the reduction of ratio graphs to the minimal mean cycle problem.

Lemma 2. *Let $G_R = ((S, s_0, E), w_1, w_2)$ be a ratio game with maximal weight W. Given a ratio $\frac{a}{b}$ with $0 \leq a \leq |S| \cdot W$ and $0 < b \leq |S| \cdot W$, we can decide whether a state has value $v = \frac{a}{b}$, $v < \frac{a}{b}$, or $v > \frac{a}{b}$ in $O(|S|^3 \cdot W^2 \cdot |E|)$ time.*

Proof. We reduce the decision for the ratio game to a decision for the mean payoff game $G_{\text{MP}} = ((S, s_0, E), w)$ with payoff function $w(e) = b \cdot w_1(e) - a \cdot w_2(e)$. In the following, let v_{R} ($v_{\text{R}}(\rho)$) be the value (of run ρ) in G_{R} and similarly for v_{MP}.

We show that $v_{\text{R}} \leq \frac{a}{b}$ implies $v_{\text{MP}} \leq 0$ and $v_{\text{R}} \geq \frac{a}{b}$ implies $v_{\text{MP}} \geq 0$. The decision whether $v_{\text{MP}} < 0$, $v_{\text{MP}} = 0$, or $v_{\text{MP}} > 0$ can be made in $O(|S|^2 \cdot W' \cdot |E|)$ time, where W' is the maximal weight in the mean payoff game [140]. We have $W' \leq b \cdot W \leq |S| \cdot W^2$, thus the decision for the ratio game can be made in $O(|S|^3 \cdot W^2 \cdot |E|)$ time.

Suppose $v_{\text{R}} \leq \frac{a}{b}$. We show that Player 1 can achieve a run of value at most 0 in G_{MP} and thus $v_{\text{MP}} \leq 0$. Let π_1 be a positional optimal Player-1 strategy for G_{R} and let π_2 be a positional optimal strategy for Player 2 in G_{MP}. Because both strategies are positional, $\rho(s_0, \pi_1, \pi_2)$ consists of a stem and a simple cycle, say $\rho = (e'_1, \ldots, e'_m) \cdot (e_1, \ldots, e_n)^\omega$. Note that

$$v_{\text{R}}(\rho) = \frac{\sum_{i=0}^{n} w_1(e_i)}{\sum_{i=0}^{n} w_2(e_i)} \text{ and } v_{\text{MP}}(\rho) = \frac{b \sum_{i=0}^{n} w_1(e_i) - a \sum_{i=0}^{n} w_2(e_i)}{n}.$$

Suppose $\sum_{i=0}^{n} w_1(e_i) > 0$, then, since $v_{\text{R}} \leq \frac{a}{b}$ and is thus finite, we have $\sum_{i=0}^{n} w_2(e_i) > 0$. It follows that

$$\frac{\sum_{i=0}^{n} w_1(e_i)}{\sum_{i=0}^{n} w_2(e_i)} \leq \frac{a}{b} \text{ implies } \frac{b \sum_{i=0}^{n} w_1(e_i) - a \sum_{i=0}^{n} w_2(e_i)}{n} \leq 0.$$

If $\sum_{i=0}^{n} w_1(e_i) = 0$, then

$$\frac{b \sum_{i=0}^{n} w_1(e_i) - a \sum_{i=0}^{n} w_2(e_i)}{n} = \frac{-(a \sum_{i=0}^{n} w_2(e_i))}{n} \leq 0.$$

The proof that $v_{\text{R}} \geq \frac{a}{b}$ implies that $v_{\text{MP}} \geq 0$ is similar, using an optimal strategy for Player 2 in G_{R}. □

Theorem 3. *Given a ratio game $((S, E), w_1, w_2)$ with maximal weight W, the value for every $s \in S$ can be computed in $O(|S|^3 \cdot W^2 \cdot |E| \cdot \log(|S| \cdot W))$.*

Proof. We use the decision procedure from Lemma 2 to perform a binary search on the list of possible values $V \setminus \{\infty\}$. If the ratio is greater than $|S| \cdot W$, it is infinite. There are less than $(|S| \cdot W)^2$ different ratios, thus we need at most $2 \cdot \log(|S| \cdot W)$ calls to the decision procedure. □

Given an algorithm to find the values of the game we can use the "group testing" technique from [140] to find optimal positional strategies.

Theorem 4. *Given a ratio game $((S, E), w_1, w_2)$ with maximal weight W, positional optimal strategies for both players can be found in $O(|S|^4 \cdot \log(\frac{|E|}{|S|}) \cdot |E| \cdot \log(|S| \cdot W) \cdot W^2)$.*

Proof. Take a state q with more than one outgoing edge, remove half of the edges, and calculate the values of the new game. If q still has the same value repeat the procedure, else repeat the procedure on the game graph with the other half of the outgoing edges. With this procedure we can successively eliminate the edges that do not contribute to the optimal values. After $\sum_{q \in S} \log(|E_q|)$ steps, where $|E_q|$ is the number of outgoing edges from q, there is only one outgoing edge per state left, which corresponds to a winning strategy. As $\sum_{q \in S} \log(|E_q|) \leq |S| \cdot \log(\frac{|E|}{|S|})$, the complexity of the algorithm is $O(|S|^4 \cdot \log(\frac{|E|}{|S|}) \cdot |E| \cdot \log(|S| \cdot W) \cdot W^2)$. □

All our ratio game algorithms are polynomial in the size of the game graph but pseudopolynomial in the weights. They are polynomial if $W = 1$.

2.4.3 Verifying and Synthesizing Robust Systems

This section describes the verification and synthesis algorithms for robust systems. First, we establish the correlation between the ratio in Definition 8 and k-robustness.

Any error specification C with cost functions c_e and c_s can be translated into a ratio game G. The weight functions w_1 and w_2 are given by the cost functions c_s and c_e respectively. Formally,

- $w_1((s_{(q,\sigma_i)}, s_{(q,\sigma_o)})) = 0$,
- $w_1((s_{(q,\sigma_o)}, s_{(q',\sigma_i)})) = c_s(q, \sigma_o \cup \sigma_i)$,
- $w_2((s_{(q,\sigma_i)}, s_{(q,\sigma_o)})) = 0$,
- $w_2((s_{(q,\sigma_o)}, s_{(q',\sigma_i)})) = c_e(q, \sigma_o \cup \sigma_i)$,

where $(s_{(q,\sigma_i)}, s_{(q,\sigma_o)}) \in E_1$ and $(s_{(q,\sigma_o)}, s_{(q',\sigma_i)}) \in E_2$ (see Section 2.3.6). Every play $\rho_G = s_0, s_{(q_0,\sigma_o)}, s_{(q',\sigma_i)}, s_{(q',\sigma'_o)} \dots$ of G corresponds to a run $\rho_C = q_0 q' \dots$ of C on $w = (\sigma_o, \sigma_i)(\sigma'_o, \sigma'_i)$.

Lemma 5. *Given a Moore machine M and an error specification C with cost function c_e and c_s, M is k-robust iff for all words $w \in L(M)$, the run $\rho(w) = q_0 \dots$ of C on $w = w_0 \dots$ satisfies*

$$v(w) = \lim_{m \to \infty} \limsup_{l \to \infty} \frac{\sum_{i=m}^{l} c_s(q_i, w_i)}{1 + \sum_{i=m}^{l} c_e(q_i, w_i)} \leq k. \tag{2.2}$$

Proof. Suppose M is k-robust. There exists a $d \in \mathbb{N}$ for any arbitrary word $w \in L(M)$ such that for all finite prefixes $w' = w_0 \dots w_n$ of w we have

$$\sum_{i=0}^{n} c_s(q_i, w_i) \leq k \cdot \sum_{i=0}^{n} c_e(q_i, w_i) + d.$$

The right hand side of the above inequality is equivalent to

$$k \cdot (1 + \sum_{i=0}^{n} c_e(q_i, w_i)) + d - k,$$

thus the above inequality can also be formulated as

$$\frac{\sum_{i=0}^{n} c_s(q_i, w_i)}{1 + \sum_{i=0}^{n} c_e(q_i, w_i)} \leq k + \frac{d-k}{1 + \sum_{i=0}^{n} c_e(q_i, w_i)}.$$

Adding $\lim_{m\to\infty} \limsup_{l\to\infty}$ on both sides of the inequality results in

$$\lim_{m\to\infty} \limsup_{l\to\infty} \frac{\sum_{i=m}^{l} c_s(q_i, w_i)}{1 + \sum_{i=m}^{l} c_e(q_i, w_i)} \leq \ k.$$

To show the last step we consider two cases, $\limsup_{l\to\infty} \sum_{i=0}^{l} c_e(q_i, w_i)$ is either some finite value d' or infinite. In the first case, $\sum_{i=0}^{n} c_s(q_i, w_i) \leq k \cdot d' + d$ for any $n \geq 0$ because M is k-robust. If $\sum_{i=0}^{n} c_s(q_i, w_i)$ is finite for any $n \geq 0$, then $\limsup_{l\to\infty} \sum_{i=0}^{l} c_s(q_i, w_i)$ is also finite. Since both $\limsup_{l\to\infty} \sum_{i=0}^{l} c_e(q_i, w_i)$ and $\limsup_{l\to\infty} \sum_{i=0}^{l} c_s(q_i, w_i)$ are finite,

$$\lim_{m\to\infty} \limsup_{l\to\infty} \frac{\sum_{i=m}^{l} c_s(q_i, w_i)}{1 + \sum_{i=m}^{n} c_e(q_i, w_i)} = 0 \leq k.$$

In the second case, the term added to k on the right hand side of the inequality is 0 in the limit:

$$\lim_{m\to\infty} \limsup_{l\to\infty} \frac{d-k}{1 + \sum_{i=m}^{l} c_e(q_i, w_i)} = 0.$$

For the other direction, consider the product CM of C and M. Then, for all $w \in L^*(M)$, $C_e(w) = CM_e(w) = \sum_{i=0}^{|w|-1} c_e(q_i, w_i)$ and $C_s(w) = CM_s(w) = \sum_{i=0}^{|w|-1} c_s(q_i, w_i)$, where $\rho_{CM}(w) = q_0 \dots q_{|w|}$ is the run of CM on w. Consider an arbitrary finite word $w \in L^*(M)$, if $|w| \leq |C| \cdot |M|$, then $CM_s(w) \leq |C| \cdot |M| \cdot W$ and $C_s(w) \leq k \cdot C_e(w) + d$ holds for any $k \geq 0$ and $d = |C| \cdot |M| \cdot W$. Otherwise, if $|w| \geq |C| \cdot |M|$, we can decompose the run $\rho_{CM}(w)$ into simple cycles $c_1, \dots, c_m$ and a simple path p consisting of the remaining nodes. (See proof of Lemma 1.) Now consider the infinite words $u_1, \dots, u_m$ that correspond to the runs leading to the cycles $c_1, \dots, c_m$, respectively, and looping there forever. We know that $u_1, \dots, u_m$ are in $L(M)$ and, due to Eq. 2.2, that $v(u_j) \leq k$ for all $1 \leq j \leq m$. Therefore, for every cycle, the sums of the weights c_e and c_s in the cycle, are either both 0 or their ratio is smaller or equal to k. Let $k = \frac{a}{b}$ and let $\frac{a_1}{b_1}, \dots, \frac{a_m}{b_m}$ be the ratios of the cycles whose ratio is nonzero, then $\sum_{i=0}^{|w-1|} c_s(q_i, w_i) = d' + \sum_{j=1}^{m} d_j \cdot a_j$ and $\sum_{i=0}^{|w-1|} c_e(q_i, w_i) = d'' + \sum_{j=1}^{m} d_j \cdot b_j$ for some $0 \leq d', d'' \leq |C| \cdot |M| \cdot W$ and $1 \leq d_j \leq |C| \cdot |M| \cdot W$. Using the fact that, if for all $1 \leq j \leq m$, $\frac{a_j}{b_j} \leq \frac{a}{b}$ holds then $\sum_{j=1}^{m} d_j \cdot a_j \leq \frac{a}{b} \sum_{j=1}^{m} d_j \cdot b_j$ holds, we obtain $\sum_{i=0}^{|w-1|} c_s(q_i, w_i) \leq \frac{a}{b} \sum_{i=0}^{|w-1|} c_e(q_i, w_i) + d'$, which proves that M is k-robust. □

Verification

We show that any robust system is k-robust.

Theorem 6. *If a Moore machine M with n_M states is robust with respect to an error specification C with n_C states and maximal system cost W, then M is $(n_C \cdot n_M \cdot W)$-robust.*

Proof. Let CM be the product of C and M. Lemma 5 shows that M is k robust if the ratio of all runs in CM is smaller or equal to k in the limit. Since one-player ratio games are positional (Lemma 1), the largest ratio corresponds to the largest ratio of a simple cycle in CM, which cannot be larger than $n_C \cdot n_M \cdot W$ because M is robust. □

Next, we show how to verify if a given Moore machine is robust or k-robust.

Theorem 7. *Given a Moore machine M with n_M states, and an error specification C over the alphabet Σ, with n_C states and maximal cost W, we can decide if M is robust in $O(n_C \cdot n_M \cdot \Sigma)$ time. Given a k, we can check if M is k-robust in $O(n_C^3 \cdot n_M^3 \cdot \Sigma)$ time.*

Proof. Let CM be the product of C and M. M is not robust iff CM contains a cycle that contains an edge with nonzero system cost and no edge with nonzero environment cost. This can be checked in time linear in the number of edges in CM, which is $n_C \cdot n_M \cdot \Sigma$. We have that M is k-robust if the maximum simple cycle ratio in CM is smaller or equal to k. The maximum simple cycle ratio in a graph with n states and m transitions can be found in $O(n^2 \cdot m)$ time [55], thus we can find the maximum ratio in $O(n_C^3 \cdot n_M^3 \cdot \Sigma)$ time. □

Synthesis

Next we show how to use Streett games to synthesize (strictly) realizing and robust systems and how to use ratio games to synthesize (strictly) realizing k-robust systems with optimal k.

Lemma 8. *Given an error specification C with n states and alphabet Σ, we can decide if a robust system exists in $O(n^2 \cdot \Sigma)$ time. If a robust system exists, it can be synthesized in $O(n^2 \cdot \Sigma)$ time.*

Proof. We translate the specification into a one-pair Streett game, F_1 is the set of states with incoming transitions with system costs and F_2 is the set of states with incoming transitions with environment costs. One-pair Streett games can be solved in $O(n \cdot m)$, where n is the number of states and m is the number of transitions [118]. □

Theorem 9. *Given an error specification C with n states and alphabet Σ, we can decide if a robust and (strictly) realizing system exists in $O(n^2 \cdot \Sigma)$ time. The system can be synthesized in $O(n^2 \cdot \Sigma)$ time.*

Proof. In order to decide if a robust and realizing system exists, build the product automaton $CA_1 = (Q \times \{q_1, q_2, q_3\}, q_0, \delta, c)$ of the error specification C and the automaton A_1 shown in Figure 2.5. Let CA_1' be CA_1, where the system costs of all transitions corresponding to the loop on state q_2 of A_1 in Figure 2.5 are

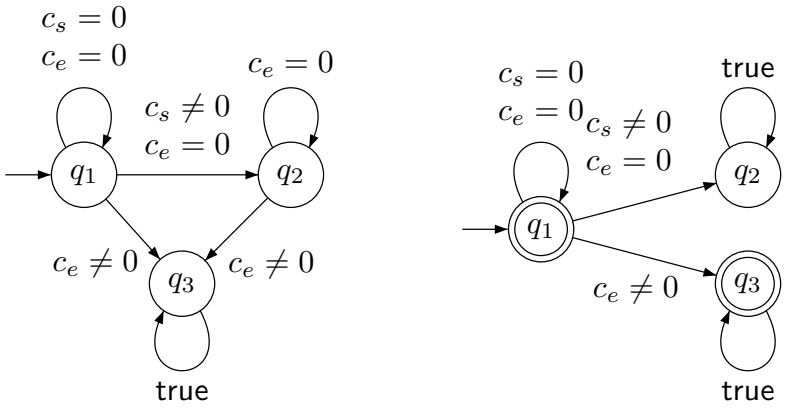


Figure 2.5: Automata A_1 and A_2 for calculating realizability and strict realizability.[3]

set to 1. Formally, the cost function of CA'_1 is $c'((q,x),\sigma) = (1, c_e((q,x),\sigma))$ if $x = q_2$ and $\delta((q,x),\sigma) = q_2$, and $c'((q,x),\sigma) = c((q,x),\sigma)$ in all other cases. Next, translate CA'_1 into a Streett game as above (proof of Lemma 8). We claim that a robust and realizing system exists iff the game is winning, and the winning strategy corresponds to a robust and realizing system.

First, assume there exists a winning strategy. No play in which Player 1 plays optimally visits a q_2-state infinitely often, because such a play has an infinite system error and zero environment error. Consequently, all words $w = (\sigma_o, \sigma_i)(\sigma'_o, \sigma'_i)\ldots$ associated with a play $\rho = s_0, s_{(q_0,\sigma_o)}, s_{(q',\sigma_i)}, s_{(q',\sigma'_o)}\ldots$ where Player 1 plays optimally satisfy $C_e(w) = 0$ implies $C_s(w) = 0$. Thus, the Moore machine corresponding to the winning strategy realizes the error specification. Second, assume there exists no winning strategy. A play where Player 2 plays optimally, has a finite environment cost and an infinite system cost. Either there exists no robust system or the play visits a q_2-state infinitely often. In the second case no system realizes the specification.

Similarly, to check for a robust and strictly realizing system, we build a Streett game from the product automaton CA'_2 of C and the automaton A_2 of Figure 2.5, where the system costs of all transitions corresponding to the loop on state q_2 are replaced by 1 and their environment costs are set to 0. Then, again any Player-1 optimal play avoids q_2-states. Consequently, for all words associated with a play where Player 1 plays optimally, all finite prefixes w' satisfy $C_e(w'[..|w'|-1]) = 0$ implies $C_s(w') = 0$. Thus, the Moore machine corresponding to a winning strategy strictly realizes the error specification. □

Lemma 10. *Given an error specification C with n states, input alphabet Σ_I, output alphabet Σ_O, and maximal cost W, if a robust system exists, a k-robust*

[3]The transitions in the automata are annotated with guards, meaning that the transition is taken if the condition in the guard evaluates to true.

system with minimal k can be synthesized in $O(n^5 \cdot (|\Sigma_I|+|\Sigma_O|)^4 \cdot \log(\frac{(|\Sigma_O|+n\cdot|\Sigma_I|)}{|\Sigma_I|+|\Sigma_O|}) \cdot (|\Sigma_O| + n \cdot |\Sigma_I|) \cdot \log(n \cdot (|\Sigma_I| + |\Sigma_O|) \cdot W) \cdot W^2)$.

Proof. We synthesize k-robust systems with ratio games. The game graph is constructed from the double cost automaton C (see Section 2.3.6). Lemma 5 shows that a positional strategy with value k corresponds to a k-robust Moore machine. An optimal positional strategy corresponds to a k-robust system with smallest possible k and $d \leq |C| \cdot W$.

The number of states in the game graph is $n \cdot |\Sigma_I| + n \cdot |\Sigma_O|$, the number of edges is $|E_1| + |E_2|$, where $|E_1| = n \cdot |\Sigma_O|$ and $|E_2| = n \cdot n \cdot |\Sigma_I|$. A winning strategy for Player 1 can be found in $O(n^4 \cdot (|\Sigma_I| + |\Sigma_O|)^4 \cdot \log(\frac{n\cdot(|\Sigma_O|+n\cdot|\Sigma_I|)}{n\cdot(|\Sigma_I|+|\Sigma_O|)}) \cdot n \cdot (|\Sigma_O| + n \cdot |\Sigma_I|) \cdot \log(n \cdot (|\Sigma_I| + |\Sigma_O|) \cdot W) \cdot W^2)$. □

Theorem 11. *Given an error specification C with n states, input alphabet Σ_I, output alphabet Σ_O, and maximal cost W, if a robust and (strictly) realizing system exists, a k-robust system with minimal k that (strictly) realizes the specification can be synthesized in* $O(n^5 \cdot (|\Sigma_I| + |\Sigma_O|)^4 \cdot \log(\frac{(|\Sigma_O|+n\cdot|\Sigma_I|)}{|\Sigma_I|+|\Sigma_O|}) \cdot (|\Sigma_O| + n \cdot |\Sigma_I|) \cdot \log(n \cdot (|\Sigma_I| + |\Sigma_O|) \cdot W) \cdot W^2)$.

Proof. For realizability translate CA'_1 from the proof of Theorem 9 into a ratio game. The system cost 1 for q_2-states guarantees that for any word w with $C_s(w) \neq 0$ and $C_e(w) = 0$ the ratio of the corresponding run has value ∞ in the ratio game. The ratios of other plays are not changed. If a play visits a q_2-state finitely often, the ratio is not influenced because we only look at the ratio in the limit.

For strict realizability translate CA'_2 from the proof of Theorem 9 into a ratio game. Since q_2-states have system cost 1 and environment cost 0, any run with a system failure before an environment failure has value ∞ in the ratio game.

A Moore machine corresponding to an optimal strategy of Player 1 is robust and (strictly) realizes the error specification. If k is the value of the initial state then M is k-robust. □

Synthesizing from Reset Error Specifications

As shown in Example 2 *reset error specifications* are an intuitive kind of error specification. We show here that every realizable reset error specification can be realized by a 1-robust Moore machine.

Definition 9. A *reset error specification* is a double cost automaton with maximal cost 1, such that for all transitions (q, σ) with $c_e(q, \sigma) = 1$ or $c_s(q, \sigma) = 1$ the next state is $\delta(q, \sigma) = q_0$.

Theorem 12. *Given a realizable reset error specification C, a 1-robust system can be synthesized in linear time.*

Proof. Translate C into a ratio game with a linear blowup, as in Lemma 10. We show that for an optimal strategy the ratio is not greater than 1. Let π_1 be

a strategy such that for all resulting plays $\rho = q_0 q_1 \ldots$, $\sum_{i=0}^{\infty} c_e(q_i, q_{i+1}) = 0$ implies $\sum_{i=0}^{\infty} c_s(q_i, q_{i+1}) = 0$. Thus, the system will not incur a cost from any state reachable using π_1 without environment cost. The only time a system cost may be incurred is when the environment incurs a cost of 1, in which case the system may also incur cost 1 and the system returns to the initial state. □

2.5 Liveness

In the previous section we studied the verification and synthesis of robust systems for safety specifications. In the case of safety, environment failures are immediately apparent and the difficulty is how the system can best recover from them. A violation of a liveness property, however, cannot be detected at any point in time [4]. Thus, a system that is robust to liveness failures must attempt to fulfill its guarantees under all circumstances, without knowing whether the environment satisfies the assumptions.

In this section, we define several possible notions of robustness in the presence of liveness, all aiming at maximizing the set of guarantees that is fulfilled for any set of fulfilled assumptions. Suppose a specification has two assumptions and two guarantees. In order for the specification to hold, both guarantees must be met when both assumptions are. A system that meets both guarantees when only one assumption is met is more robust than one that meets one (or zero) guarantees when only one assumption is met.

Example 3. We consider a variant of the dining philosophers problem [58]. There are n philosophers sitting at a round table. There is one chopstick between each pair of adjacent philosophers. Because each philosopher needs two chopsticks to eat, adjacent philosophers cannot eat simultaneously. We are interested in schedulers that use input variables h_i signifying that philosopher i is hungry and output variables e_i signifying that philosopher i is eating.

We have the following requirements. First, an eating philosopher prevents her neighbors from eating. Formally, $G_{1i} = \mathsf{G}(e_i \rightarrow \neg e_{(i-1) \bmod n} \wedge \neg e_{(i+1) \bmod n})$. Second, an eating philosopher eats until she is no longer hungry: $G_{2i} = \mathsf{G}(e_i \wedge h_i \rightarrow \mathsf{X}\, e_i)$. Third, every hungry philosopher eats eventually $G_{3i} = \mathsf{G}(h_i \rightarrow \mathsf{F}\, e_i)$. We add the assumption that an eating philosopher eventually loses her appetite: $A_{1i} = \mathsf{G}(e_i \rightarrow \mathsf{F}\, \neg h_i)$. Our final specification consists of n assumptions and $3n$ guarantees: $\bigwedge_{i=1}^{n} A_{1i} \rightarrow \bigwedge_{i=1}^{n} (G_{1i} \wedge G_{2i} \wedge G_{3i})$.

We have synthesized a system realizing this specification for 5 philosophers using our synthesis tool RATSY[4]. The system constructed by RATSY is not very robust: When philosopher 1 violates the assumption by always being hungry, then philosophers 1 and 3 eat forever, while the other philosophers starve. Thus the three guarantees $\mathsf{G}(h_2 \rightarrow \mathsf{F}\, e_2)$, $\mathsf{G}(h_4 \rightarrow \mathsf{F}\, e_4)$, and $\mathsf{G}(h_5 \rightarrow \mathsf{F}\, e_5)$ are violated. A more robust system would let philosopher 3 and 4 take turns, thus violating only two guarantees. □

[4] `http://rat.fbk.eu/ratsy/index.php/Main/HomePage`

In this section we consider GR(1) specifications. GR(1) is an expressive specification formalism with a natural distinction between assumptions and guarantees [119]. Efficient tools exist for GR(1) specifications, which have been used to synthesize relatively large specifications [95, 26]. GR(1) specifications are of the form $\varphi \to \psi$. Here, φ represents the environment assumptions and ψ represents the system guarantees and both φ and ψ are given as a set of deterministic Büchi automata. These automata are combined into a product automaton with state space Q, transition relation δ, and acceptance condition $\bigwedge_{i=1}^{m} \mathsf{GF}\, a_i \to \bigwedge_{i=1}^{n} \mathsf{GF}\, g_i$.

GR(1) specifications do not require any guarantees to be fulfilled when some assumption is violated. We propose an intuitive notion of robustness that prescribes, for any number of environment assumptions that is violated, a minimal number of system guarantees that must still be fulfilled. We show that this and related measures of robustness can be transformed to a specification of the form $\bigwedge_{j=1}^{k}(\bigwedge_{i=1}^{m} \mathsf{GF}\, a_{ji} \to \bigwedge_{i=1}^{n} \mathsf{GF}\, g_{ji})$, which is a Generalized Reactivity (generalized Streett) formula of rank k. We address the problem of verification and especially of synthesis of such formulas, which allows us to construct robust systems.

The verification problem is a relatively straightforward generalization of the verification problem for GR(1) (cf. [83]) and can be performed in time $O(m \cdot n \cdot |Q| \cdot |\delta|)$. Recall that m is the number of assumptions and n is the number of guarantees, and $|Q|$ and $|\delta|$ refer respectively to the size of the state space and the transition relation of the product automaton.

The synthesis question is answered by solving a Generalized Reactivity game. This can either be done through a specialization of Zielonka's algorithm, or through a novel algorithm presented in this work, both of which can be implemented symbolically. Zielonka's algorithm runs in time $O(|Q|^{2 \cdot k} \cdot |\delta| \cdot (m+n)^k \cdot k!)$, which we improve to $O(|Q|^k \cdot |\delta| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$. On the other hand, our algorithm produces larger strategies and thus larger robust systems: the systems produced by Zielonka's algorithm have size $|Q| \cdot n^k \cdot k!$, whereas our algorithm produces systems of size $|Q| \cdot ((m+1) \cdot (n+1))^k \cdot k!$.

Our algorithm is a generalization of a game-theoretic algorithm for the important class of GR(1) conditions based on a reduction (via a counting construction) to Streett games with a single pair. The algorithm runs in time $O(|Q| \cdot |\delta| \cdot (m \cdot n)^2)$. This bound improves the $O(|Q|^2 \cdot |\delta| \cdot m \cdot n)$ time bound of the algorithm of [119] for the case that Q is larger than m and n, which is typical in such applications as GR(1) synthesis.

This part of the thesis is structured as follows. First we define several notions of robustness in Section 2.5.1. In order to solve the synthesis problem for robust systems, we introduce the necessary transformations on the formulas and game theoretic algorithms in Sections 2.5.2 and 2.5.3. In Section 2.5.4 we return to the questions of verification and synthesis of robust systems.

2.5.1 Defining Measures of Robustness

In this section we discuss how to compare systems with respect to robustness. Usually, multiple systems satisfy a specification, but which one is most robust? In Section 2.4 we answered this question for safety specifications: our measure of robustness for a safety specification $\varphi \rightarrow \psi$ is the ratio between how often the environment violates φ and how often the system violates ψ. For specifications with liveness properties, this approach does not work because we cannot count the number of violations of a liveness property. Instead, we propose to count the number of properties violated. In the following we show two different robustness measures, the single and the multiple counting requirements measure. Then we formally state the requirements a robustness measure has to satisfy.

Single Counting Requirements

Recall the dining philosophers example with $n = 5$ philosophers given in the introduction. Suppose system D_1 always lets one philosopher eat until she is not hungry anymore and then moves to the next hungry philosopher in a round robin manner. If one philosopher is hungry forever, then no other philosopher gets to eat again. Thus, the violation of one assumption leads to the violation of four guarantees.

Suppose system D_2 lets two non-adjacent philosophers eat at the same time until neither is hungry anymore. They take turns in the following order: first philosopher 1 and 3 eat, then philosopher 2 and 4, and last philosopher 3 and 5 eat. If one of the currently eating philosopher is hungry forever, then the two currently eating philosophers eat forever and no other philosopher gets to eat again. Thus, the violation of one assumption leads to the violation of three guarantees. System D_2 is thus more robust than system D_1.

An even more robust system (D_3) is the one described in the introduction. Two philosophers eat at the same time, as soon as one of them is not hungry anymore another philosopher with free chopsticks is allowed to eat. If one philosopher is hungry forever, she eats forever and the other philosophers that are not her neighbors take turns eating. The violation of one assumption leads to the violation of two guarantees.

We specify robust systems by adding restrictions to the original specification. All three systems above satisfy the original specification $\varphi = \bigwedge_{i=1}^{n} A_{1i} \rightarrow \bigwedge_{i=1}^{n}(G_{1i} \wedge G_{2i} \wedge G_{3i})$, but only D_2 and D_3 guarantee that they violate at most three system guarantees if the environment violates one of its assumptions. Formally, D_2 and D_3 additionally satisfy

$$\psi_1 = \Big(\bigvee_{i=1}^{n} \bigwedge_{j \in \{1,\dots,n\} \setminus \{i\}} A_{1j}\Big) \rightarrow \Big(\varphi_S \wedge \bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} \bigvee_{k=j+1}^{n} \bigwedge_{l \in \{1,\dots,n\} \setminus \{i,j,k\}} G_{3l}\Big),$$

where $\varphi_S = \bigwedge_{i=1}^{n}(G_{1i} \wedge G_{2i})$. The antecedent of the formula states that the environment satisfies $n-1$ out of the n assumptions. The consequent says that the system satisfies all the safety guarantees (G_{1i} and G_{2i}) but might violate three of its liveness guarantees.

Note that in general, a robust system cannot violate a safety guarantee in response to a violation of a fairness assumption, since a violation of a fairness assumption can not be detected in finite time.

Since D_3 violates at most two system guarantees if one environment assumption is violated, it also satisfies the following formula.

$$\psi_2 = \Big(\bigvee_{i=1}^{n} \bigwedge_{j\in\{1,\dots,n\}\setminus\{i\}} A_{1j}\Big) \rightarrow \Big(\varphi_S \wedge \bigvee_{i=1}^{n}\bigvee_{j=i+1}^{n} \bigwedge_{k\in\{1,\dots,n\}\setminus\{i,j\}} G_{3k}\Big)$$

These two formulas allow us to distinguish between systems D_1, D_2, and D_3, which satisfy the same base specification but differ in how resilient they are with respect to violated environment assumptions. We propose to use formulas of this type, which relate the number of satisfied assumptions to a number of satisfied guarantees to measure how robust a system is.

Suppose $\mathcal{A}$ is a set of assumptions and $\mathcal{G}$ is a set of guarantees. Let $\mathcal{A}_k = \{A \subseteq \mathcal{A} \mid |A| = k\}$ be the set of all subsets of $\mathcal{A}$ of size k and let $\mathcal{G}_k$ be defined similarly. We can augment the specification with a restriction of the form $(\bigvee_{A\in\mathcal{A}_k} \bigwedge_{A_i\in A} A_i) \rightarrow (\bigvee_{G\in G_l} \bigwedge_{G_i\in G} G_i)$ to check if a system satisfies l guarantees when k assumptions are satisfied. Naturally, a system that satisfies more guarantees with the same number of satisfied assumptions is more robust.

Multiple Counting Requirements

In some cases we might want to have a more fine-grained measure of robustness, which cannot be expressed by a single restriction of the form given above. Recall again the dining philosophers example but this time assume there are $n = 7$ philosophers. Suppose system D_4 allows two hungry philosophers to eat at the same time. Then, even if one philosopher does not stop eating, the other non-adjacent philosophers can still take turns eating. However, if two philosophers misbehave and they both get to eat (i.e., they do not sit next to each other), they will leave the other five philosophers to starve. Suppose another system D_5 allows three philosophers to eat at the same time. Now, if two philosophers misbehave and they both get to eat, the system D_5 still allows another philosopher to eat and only four philosophers are left to starve. Both D_4 and D_5 realize the specification φ. If we consider the restrictions from above, we see that both systems satisfy the formula ψ_1 and ψ_2. Our previous measure of robustness cannot distinguish between D_4 and D_5. Let's add another restriction ψ_3 to our specification:

$$\psi_3 = \Big(\bigvee_{i=1}^{n}\bigvee_{j=i+1}^{n} \bigwedge_{k\in\{1,\dots,n\}\setminus\{i,j\}} A_{1k}\Big) \rightarrow \Big(\varphi_S \wedge \bigvee_{i=1}^{n}\bigvee_{j=i+1}^{n}\bigvee_{k=j+1}^{n} \bigwedge_{l\in\{1,\dots,n\}\setminus\{i,j,k\}} G_{3l}\Big)$$

System D_5 realizes $\varphi \wedge \psi_2 \wedge \psi_3$ but system D_4 does not. We can measure the number of satisfied guarantees for several numbers of satisfied assumptions. The restrictions we add to the specifications are of the form

$$\bigwedge_{(k,l)\in L} \Big(\Big(\bigvee_{\mathcal{A}\in\mathcal{A}_k} \bigwedge_{A_i\in\mathcal{A}} A_i\Big) \rightarrow \Big(\bigvee_{\mathcal{G}\in\mathcal{G}_l} \bigwedge_{G_i\in\mathcal{G}} G_i\Big)\Big)$$

where L is a list of pairs (k, l), requiring l guarantees to be satisfied if k assumptions are satisfied.

Definitions

Both single and multiple counting requirements, as defined above, can be put in the following form (as we will shown in Section 2.5.2).

Definition 10. Given a GR(1) specification $A^{GR(1)}$ with a set of m assumptions $\{J_1^a, \ldots, J_m^a\}$ and a set of n guarantees $\{J_1^g, \ldots, J_n^g\}$, a *robustness specification* for $A^{GR(1)}$ has the form

$$\bigwedge_{l=1}^{k} \left(\bigwedge_{i=1}^{m_l} \mathcal{B}(J_{l,i}^a) \rightarrow \bigwedge_{i=1}^{n_l} \mathcal{B}(J_{l,i}^g) \right),$$

where $J_{l,i}^a \in \{J_1^a, \ldots, J_m^a\}$ and $J_{l,i}^g \in \{J_1^g, \ldots, J_n^g\}$.

There is a natural partial order on robustness specifications: If, for each set of satisfied assumptions, a specification S requires a superset of the guarantees required by specification S', then S is more robust than S'. Let us denote this order by $\prec$.

Definition 11. A *robustness measure* for a GR(1) specification is a set of robustness specifications together with a total order that respects $\prec$.

For example, consider again the 'simple counting requirements' robustness specifications from above. A possible total order is $(k = 0, l = |\mathcal{G}|) > (k = 0, l = |\mathcal{G}| - 1) > \ldots > (k = 0, l = 1) > (k = 1, l = |\mathcal{G}|) > \ldots > (k = |\mathcal{A}|, l = 0)$, where k is the number of satisfied assumptions and l the number of satisfied guarantees. Another possible total order is $(k = 0, l = |\mathcal{G}|) > (k = 1, l = |\mathcal{G}|) > \ldots > (k = |\mathcal{A}| - 1, l = |\mathcal{G}|) > (k = 0, l = |\mathcal{G}| - 1) > \ldots > (k = |\mathcal{A}|, l = 0)$. A total order is necessary to synthesize the most robust specification.

Section 2.5.4 shows how to verify and synthesize robust systems for a given measure. To synthesize a robust system, we solve games with the robustness specification as objective. Section 2.5.3 shows how to solve such games. In the next section, we show how to translate combinations of Büchi objectives to generalized Büchi objectives.

2.5.2 Simplifying Combinations of Büchi Objectives

In this section we present a simplification of Disjunctions of Conjunctions of Büchi objectives (DCB objective) to conjunctions of Büchi objectives (generalized Büchi objectives). This simplification is needed to transform counting requirements to robustness specifications. The simplification (or reduction) incurs an exponential blowup. Games with generalized Büchi objectives can be solved in polynomial time, whereas we show that games with DCB objectives are coNP-complete. This shows that the exponential blow up in the translation is probably inevitable.

Simplification of DCB objectives

The simplification is done in two steps. First, we show how to translate DCB objectives to conjunctions of disjunctions of Büchi objectives. Second, we show that conjunctions of disjunctions of Büchi objectives can be translated to generalized Büchi objectives.

Lemma 13. *Any winning condition ψ that is a DCB objective can be translated into an equivalent winning condition ψ' that is a conjunction of disjunctions of Büchi objectives, such that $|\psi'| = O(2^{|\psi|})$.*

Proof. For any objective $\psi = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} \mathcal{B}(B_{ij})$ there exists an equivalent objective $\psi' = \bigwedge_{i=1}^{n^m} \bigvee_{j=1}^{m} \mathcal{B}(B'_{ij})$ with $B'_{ij} \in \{B_{ij} \mid i \in \{1 \dots m\} \text{ and } j \in \{1 \dots n\}\}$. The translation is identical to that of changing Disjunctive Normal Form (DNF) into Conjunctive Normal Form (CNF). □

Lemma 14. *Any winning condition ψ that is a conjunction of disjunctions of Büchi objectives can be translated into an equivalent generalized Büchi objective ψ', such that $|\psi'| = O(|\psi|)$.*

Proof. Because a disjunction of Büchi conditions is again a Büchi condition $(\mathcal{B}(B_1) \vee \mathcal{B}(B_2) = \mathcal{B}(B_1 \cup B_2))$, objectives of the form $\bigwedge_{i=1}^{k} \bigvee_{j=1}^{l} \mathcal{B}(B_{ij})$ can be reduced to a generalized Büchi objective $\bigwedge_{i=1}^{k} \mathcal{B}(\bigcup_{j=1}^{l} B_{ij})$. □

Corollary 15. *Any winning condition ψ that is a DCB objective can be translated into an equivalent generalized Büchi objective ψ', such that $|\psi'| = O(2^{|\psi|})$.*

Complexity of solving DCB objectives

We first show that the problem of deciding whether Player 1 has a winning strategy for a DCB objective is coNP-hard, and then we will argue coNP-completeness.

Lemma 16. *Given a game graph with a DCB objective, deciding if Player 1 has a winning strategy is coNP-hard.*

Proof. We show that the problem of deciding whether Player 1 has a winning strategy in a game with a DCB objective is at least as hard as deciding whether a 3SAT formula is unsatisfiable. Consider a 3SAT formula ψ in CNF with clauses $C_1, C_2, \dots, C_k$ over variables $\{x_1, x_2, \dots, x_n\}$, where each clause consists of disjunctions of exactly three literals (a literal is a variable or its complement). Given the formula ψ, we construct a game graph as shown in Figure 2.6. The game graph is as follows: from the initial state, Player 1 chooses a clause, then from a clause Player 2 chooses a literal that appears in the clause (i.e., makes the clause true). From every literal the next state is the initial state. The winning condition for Player 1 is $\bigvee_{i=1}^{n}(\mathcal{B}(X_i) \wedge \mathcal{B}(\overline{X_i}))$, where X_i is the set of states that correspond to the literal x_i and $\overline{X_i}$ is the set of states that correspond to the complement literal $\neg x_i$; in other words, Player 1 wants to visit some variable and its complement infinitely often.

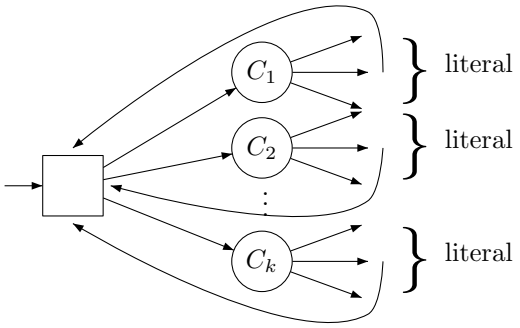


Figure 2.6: Game graph for 3SAT formula

We now present two directions of the hardness proof.

Not satisfiable implies winning. We show that if ψ is not satisfiable, then Player 1 has a winning strategy. The winning strategy is as follows: the strategy is played in rounds; in round i Player 1 chooses the clauses $C_1, C_2, \ldots, C_k$ in order, and then proceeds to round $i+1$. Since ψ is not satisfiable, for every round i there is at least one variable such that both the variable state and its complement state is visited in round i. Since the number of variables is finite, it follows that there must be some variable such that both the variable state and its complement state appears infinitely often. The result follows.

Satisfiable implies not winning. We now show that if ψ is satisfiable, then Player 2 has a winning strategy. Consider a satisfying assignment to ψ. A memoryless winning strategy for Player 2 is as follows: for every clause C_i, Player 2 chooses a literal from C_i that is set true by the satisfying assignment. Given the strategy of Player 2, since the strategy is obtained from a valid assignment, it follows that never a variable and its complement is visited. □

Lemma 17. *Given a game graph with a DCB objective, deciding if Player 1 has a winning strategy can be achieved in coNP.*

Proof. The proof is as follows: we have already shown that DCB objectives can be translated to a generalized Büchi objective (which is an upward-closed objective). It follows from the result of Zielonka [139] that there are memoryless winning strategies for the complement of an upward-closed objective (in particular for disjunction of coBüchi objectives). It follows that there always exist memoryless winning strategies for Player 2. Hence to falsify that Player 1 has a winning strategy, a memoryless strategy for Player 2 can be fixed (as the polynomial witness) and the resulting one-player graph can be verified in polynomial time. The polynomial time verification procedure uses the following fact: consider a Maximal Strongly Connected Component (MSCC) in a one-player graph (only Player 1), then the MSCC is winning if for some index i of the disjunction, for every index j of the corresponding conjunction the MSCC contains at least one Büchi state B_{ij}. Using the above fact, MSCC decomposition of a graph,

and reachability to winning MSCCs we obtain a polynomial time verification procedure. □

Lemma 16 and Lemma 17 yield the following result.

Theorem 18. *Given a game graph with a DCB objective for Player 1, deciding if Player 1 has winning strategy is co-NP complete.*

2.5.3 Solving Generalized Reactivity Games

In this section, we first present a translation of GR(1) winning conditions to one-pair Streett conditions (or parity $\{0,1,2\}$ conditions). Our reduction is based on a counting construction similar to the reduction of generalized Büchi conditions to Büchi conditions. Second, we generalize the translation to reduce games with Generalized Reactivity objectives to games with Streett objectives.

Reduction

Consider a GR(1) game $G = ((S,E),\mathrm{Acc})$ with acceptance condition $\mathrm{Acc} = \bigwedge_{i=1}^{m} \mathcal{B}(A_i) \rightarrow \bigwedge_{i=1}^{n} \mathcal{B}(G_i)$, with Player 1 states S_1, and Player 2 states S_2. We construct an equivalent one-pair Streett game $G' = ((S',E'), \mathcal{B}(A'_1) \rightarrow \mathcal{B}(G'_1))$ with Player 1 states S'_1 and Player 2 states S'_2 as follows.

1. The state space $S' = S \times \{0,1,\ldots,m\} \times \{0,1,\ldots,n\}$, with $S'_1 = S_1 \times \{0,1,\ldots,m\} \times \{0,1,\ldots,n\}$, and $S'_2 = S_2 \times \{0,1,\ldots,m\} \times \{0,1,\ldots,n\}$.
2. The set of edges E' is defined as follows:

$$
\begin{array}{ll}
((s,i,n),(s,0,0)) \in E' & \text{for } 0 \leq i \leq m, \\
((s,m,j),(s,0,j)) \in E' & \text{if } j \neq n, \text{ and} \\
((s,i,j),(s',i',j')) \in E' & \text{if } (s,s') \in E, \text{ with} \\
 & \quad i' = i+1 \text{ if } s' \in A_{i+1} \text{ otherwise } i' = i, \text{ and} \\
 & \quad j' = j+1 \text{ if } s' \in G_{j+1} \text{ otherwise } j' = j.
\end{array}
$$

3. The Streett pair is $(A'_1 = \{(s,m,j) \in S' \mid j \in \{0,\ldots,n\}\}, G'_1 = \{(s,i,n) \in S' \mid i \in \{0,\ldots,m\}\})$.

We present the intuition behind the construction. Initially i and j are zero. If all the assumptions are visited such that, assumption A_2 is visited after some visit to assumption A_1; assumption A_3 is visited after some visits to assumptions A_1, A_2; assumption A_4 is visited after some visits to assumptions A_1, A_2, A_3; and so on, since the last reset, then i is reset to 0. If all the guarantees are visited, such that guarantee G_2 is visited after some visit to guarantee G_1; guarantee G_3 is visited after some visits to guarantees G_1, G_2; guarantee G_4 is visited after some visits to guarantees G_1, G_2, G_3; and so on, since the last reset, then j is reset to 0. In between resets, i and j denote the last assumption and the last guarantee visited in the order described above, since the last reset.

The size of the new state space is $|S'| = |S| \cdot (m+1) \cdot (n+1) = O(|S| \cdot m \cdot n)$. The new number of transitions is $|E'| = |E| \cdot (m+1) \cdot (n+1) + 2 \cdot |S| = O(|E| \cdot m \cdot n)$.

Lemma 19. *There exists a winning strategy for G iff there exists a winning strategy for G'.*

Proof. Consider a play ρ in G and the corresponding play ρ' in G'. We consider two cases.

1. We consider the case where all guarantees appear infinitely often in ρ. If all guarantees are visited infinitely often, then a state with third state component with value n is visited infinitely often in ρ' (i.e., G'_1 is visited infinitely often). Thus, if the play in G satisfies the GR(1) condition by visiting all guarantees infinitely often, then the corresponding play in G' visits G'_1 infinitely often and satisfies the Streett condition.

2. We consider the case where some guarantee is not visited infinitely often in ρ. In this case a state with third state component with value n is visited only finitely often in ρ'. We consider two sub-cases.

   (a) If all the assumptions are visited infinitely often in ρ, then a state with second state component with value m is visited infinitely often in ρ'. In this case the play in G does not satisfy the GR(1) condition, and the corresponding play in G' visits A'_1 infinitely often and G'_1 finitely often, which violates the Streett condition.

   (b) If some assumption is not visited infinitely often in ρ, then a state with second state component with value m is visited only finitely often in ρ' (i.e., A'_1 is visited finitely often). In this case the play in G satisfies the GR(1) condition, and the corresponding play in G' satisfies the Streett condition.

This completes the proof. □

Theorem 20. *Games with GR(1) objectives can be solved in $O(|S| \cdot |E| \cdot (m \cdot n)^2)$ time.*

Proof. One-pair Streett (or parity $\{0, 1, 2\}$)) games with $|S'|$ states and $|E'|$ edges can be solved in $O(|S'| \cdot |E'|)$ time [97]. From Lemma 19 we obtain the above theorem. □

It may also be noted that one-pair Streett games can be solved very efficiently in practice [57] and also symbolically [68] (and implementing our counting construction symbolically is standard). The previous best know algorithm to solve GR(1) games was through the triple nested fix-point algorithm of [119] which works in time $O(|S|^2 \cdot |E| \cdot n \cdot m)$. For the typical case that $|S|$ is much greater than m and n, our algorithm is faster.

Our algorithm can easily be generalized to Generalized Reactivity objectives.

Theorem 21. *Games with Generalized Reactivity objectives can be solved in $O(|S|^k \cdot |E| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$ time.*

Proof. Turn all GR(1) objectives into Streett pairs, the Streett game has $O(|S| \cdot m^k \cdot n^k)$ states, $O(|E| \cdot m^k \cdot n^k)$ transitions, and k-Streett pairs. A Streett game with k pairs, $|E'|$ transitions and $|S'|$ states can be solved in $O(|E'| \cdot |S'|^k \cdot k!)$ [118]. □

A symbolic algorithm for Generalized Reactivity objectives can be obtained as follows: use the standard symbolic implementation of the counting construction along with the symbolic algorithm for Streett games from [118]. This gives us a symbolic algorithm for solving games with Generalized Reactivity objectives.

Winning Strategy and Required Memory

A winning strategy for a GR(k) condition is obtained as follows: first we consider an automaton A_1 of size $((n+1) \cdot (m+1))^k$ to store the values of the counters and follow the transition as given in the reduction to Streett games with k pairs (essentially this mimics the reduction of the counting construction). Winning strategies in Streett games with k pairs require at most $k!$ memory, and a winning strategy (automata A_2 with $k!$ memory) can be constructed from the Streett game solving algorithms (such as [118] or [43]). The product automaton $A_1 \times A_2$ describes a winning strategy for the GR(k) condition and requires $((n+1) \cdot (m+1))^k \cdot k!$ memory.

In the case of GR(1) conditions, our construction of winning strategies requires $(n+1) \cdot (m+1)$ memory. The memory can be improved to n as follows: once the winning set is computed, we can run Zielonka's algorithm to compute a winning strategy with n memory. However, as the winning set is already computed we can get rid of the outer iteration of Zielonka's algorithm and re-running Zielonka's algorithm to compute the winning strategy, given the winning set, takes $O(|S| \cdot |E| \cdot (n+m))$ time.

2.5.4 Verifying and Synthesizing Robust Systems

First, we show how to verify whether a system has a certain level of robustness. Then, we give an algorithm to synthesize the most robust system with respect to a given robustness measure.

Verification

Verification of a robustness specification is similar to the verification of a GR(1) specification.

Lemma 22. *Given a GR(1) specification $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$ with m assumptions and n guarantees, and a system M, verification can be performed in $O(m \cdot n \cdot |Q|^2 \cdot |\delta|)$ time.*

Proof. Check if a trace in $A^{GR(1)} \times M$ satisfies $\bigwedge_{i=1}^{m} \mathcal{B}(A_i) \wedge (\bigvee_{i=1}^{n} \neg \mathcal{B}(G_i))$ (the negation of the specification) using the μ-calculus formula $\mu X.(pre(X) \vee \bigvee_{j=1}^{n} \nu Y.(\neg G_j \wedge \bigwedge_{i=1}^{m} pre(\mu Z.(Y \wedge (A_i \vee pre(Z))))))$ [83]. The complexity of the nested fix-points is in $O(|Q|^2 \cdot |\delta|)$ [69]. □

Theorem 23. *Given a GR(1) specification $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$, a robustness specification $\bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \rightarrow \bigwedge_{i=1}^{n} \mathcal{B}(G_{l,i}))$, and a system M, verifying that M satisfies the robustness specification takes $O(k \cdot m \cdot n \cdot |Q|^2 \cdot |\delta|)$ time.*

Proof. Check if a trace in $A^{GR(1)} \times M$ satisfies the negation of the specification $\bigvee_{l=1}^{k}(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \wedge (\bigvee_{i=1}^{n} \neg \mathcal{B}(G_{l,j})))$ by checking the k different GR(1) parts $(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \wedge (\bigvee_{i=1}^{n} \neg \mathcal{B}(G_{l,j})))$ separately, one after the other, using the method of Lemma 22. □

Synthesis

The most robust system with respect to a given robustness measure can be synthesized by synthesizing the greatest realizable robustness specification. Thus, synthesis can be reduced to solving GR games.

Theorem 24. *Given a GR(1) specification $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$, and a robustness measure with h robustness specifications $r_p = \bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \rightarrow \bigwedge_{i=1}^{n} \mathcal{B}(G_{l,i}))$, with $1 \leq p \leq h$, and a total order, synthesis of the most robust system can be performed in $O(h \cdot |Q|^k \cdot |\delta| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$ time. The size of the resulting system is $((m+1) \cdot (n+1))^k \cdot k! \cdot |Q|$.*

Proof. The best system can be synthesized by trying the specifications in order. Start with the largest robustness specification according to the given total order. Try to synthesize a system satisfying the specification using the algorithm given in Section 2.5.3. The translation of the specification into a game graph is linear, hence synthesis of a robustness specification can be performed in $O(|Q|^k \cdot |\delta| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$ time (see Theorem 21). The size of the synthesized system is $((m+1) \cdot (n+1))^k \cdot k! \cdot |Q|$, if the robustness specification is realizable. If the robustness specification is not realizable proceed with the next specification in the given order. □

2.6 Related Work

A large number of diverse related work exists. We look at properties related to robustness, namely continuity (Section 2.6.1) and fault tolerance (Section 2.6.2). We consider synthesis of robust controllers (Section 2.6.3) which is usually not game based, and games (Section 2.6.4). We discuss inherently robust specification formalisms (Section 2.6.5), and work on how to handle environment assumptions (Section 2.6.6). Last but not least we give a short overview of extensions of our work (Section 2.6.7).

2.6.1 Continuity

In the continuous domain, it is natural to require systems to be continuous, which guarantees robustness in the sense that a small output error can be guaranteed by an appropriately small input error [90]. This notion is not appropriate in the

discrete setting, as discrete functions are in general not continuous. Consider, for example, a specification that requires that the value of the output g is always true (false) if the initial input r is true (false, respectively): $(r \rightarrow \mathsf{G}\, g) \wedge (\neg r \rightarrow \mathsf{G}\, \neg g)$. Here, a minimal difference in the input, namely a change of the initial input, causes a maximal difference in the output.

Doyen et al. [62] solve this problem for sequential circuits by leaving out discontinuous parts of the circuit from the robustness analysis. In contrast to our definition, their notion of robustness is independent of the specification. A sequential circuit is Σ_D robust if a finite number of changes in the disturbance inputs (inputs that do not result in wanted discontinuous behavior) results in a finite number of changes in the computed outputs. They characterize the class of sequential circuits that are Σ_D robust and provide an algorithm to decide whether a sequential circuit is Σ_D robust.

Similarly, Samanta et al. [127, 91, 92] analyze how changes in input sequences incur changes in output sequences. A transducer is K-Lipschitz robust if for all input sequences with a finite distance, the resulting output sequences have a distance less than K times the distance of the input sequences. A possible distance function could be the Hamming distance which counts the number of positions in which the sequences differ. In [91] Samanta et al. consider arbitrary nondeterministic transducers and arbitrary distance functions and in [92] they extend their work to timed transducers. In comparison, our approach is based on specifications and we show how to synthesize a robust system from a robustness specification. We do not exclude discontinuous behavior but the user has to explicitly specify the wanted robustness behavior. Additionally we restrict ourselves to Moore machines.

Also a related robustness measure is the Input-Output stability defined by Tabuada et al. [131, 132, 126]. Similarly to our work in Section 2.4 the user has to specify a cost function for the inputs and the outputs respectively. One part of the Input-Output stability is the *robustness gain* which relates the output costs to the input costs. The second part of the Input-Output stability is the *rate of decay* which measures the number of steps until the transducer returns to its nominal behavior. Tabuada et al. show how to verify Input-Output stability for a given transducer and given input and output costs. They also give an algorithm to synthesize a controller such that a given transducer together with the controller is Input-Output stable for given input and output costs. The *robustness gain* is similar to our notion of robustness defined in Section 2.4. It is not clear, how the *rate of decay* can be applied to LTL synthesis.

2.6.2 Fault Tolerance

Measures of robustness for different fault models, for example internal malfunctions of circuits [74], have been studied. Classical notions of fault tolerance such as self-stabilization [59] and the notions of closure and convergence suggested in [8] focus on safety properties. Convergence requires that a system restores its invariant after an error has occurred, and closure requires that the system satisfies a second, larger invariant even when errors recur. Our approach can be

viewed as an extension of closure to liveness, where we require that some weaker set of guarantees is fulfilled when the environment behaves unexpectedly.

Attie et al. [9] argue that fault-intolerant programs are often unrealistic. They introduce a framework to specify fault-tolerant concurrent programs with CTL formulas and different levels of tolerance, and show how to synthesize such programs. Contrary to our work, this work considers closed systems and requires the developer to specify possible faults explicitly. People have also studied the problem of retrofitting fault tolerance to existing programs. (See, e.g., [101, 64, 81, 44].)

A topic closely related to fault tolerance is the topic of error resilience as considered in [66] and [138]. Ehlers and Topcu [66] show how to automatically synthesize an error resilient system from a GR(1) specification. A system is (k,b)-resilient, if the system can satisfy its guarantees, assuming that after k environment safety violations there are b steps without environment violations and the number of overall environment violations is finite. In contrast we look at the infinite behavior, and at infinite safety violations. While error resilience requires systems to continue satisfying their guarantees for a small number of environment violations, robustness deals with the environment errors that inevitably lead to system errors.

Wong et al. [138] extend the work of synthesizing error resilient systems, for robot controllers. They automatically change the environment safety assumptions if an error resilient behavior of the robot is not possible and synthesize a new robot controller for the new environment assumptions. The intuition behind this strategy is that the environment might change or that the environment was not modeled correctly. The use case we consider does not allow a change of the system at runtime (e.g if we synthesize hardware) and does not consider changing environments as might be the case for a robot motion control.

2.6.3 Controllers

In general controller synthesis and reactive systems synthesis are two different fields of research. Reactive system synthesis as considered in this work is based on games, where the environment is an uncontrolled adversary. In contrast, in classic controller synthesis no uncontrolled environment has to be considered. Controller synthesis is usually based on language inclusion. This changes when we want to synthesize a robust controller, a controller that reacts properly for unexpected or undefined behavior of the controlled discrete event system. Several notions of robustness have been defined in the field of controller synthesis. In the following we make a comparison with our notion of robustness.

Cury and Krogh [51] consider synthesis of robust controllers for discrete event systems, where a controller is optimal if it produces the correct behavior for the original plant and produces behavior between an upper and lower bound specification for a maximal set of plants. The method produces a controller that is (i) correct with respect to a given plant and a specification and (ii) correct for as many plants as possible for a slightly changed specification. Although the solution to the problem is quite different, the main idea of the robustness

definition is comparable to our synthesis approach for GR(1) specifications with a robustness measure, described in Section 2.5. We synthesize a system that satisfies the original GR(1) specification and satisfies similar guarantees with as little environment assumptions as possible.

Topcu et al. [134] define robustness to uncertainties for discrete controllers. They show how to synthesize a discrete controller such that the system satisfies a given GR(1) specification even if some of the transitions in the transition system are not known (modeled uncertainties). As for fault tolerant systems the faults (uncertainties) have to be modeled by the user. In contrast, to synthesize robust systems as described in this work, the user does not have to model the possible faults directly.

Another work on the design of robust discrete controllers is [105]. Majumdar et al. synthesize controllers from metric automata, automata with a distance measure for states. The intuition is that a strategy that chooses states which are close to each other results in behavior that satisfies related properties. Thus to synthesize a controller which is robust with respect to disturbances one has to choose a strategy with states close to the states of the original strategy. This results in behavior close to the original behavior when disturbances occur. In our work we do no define a metric on states, we let the user define an error specification or robustness measure.

D'Souza and Gopinathan [63] consider specifications for controllers that are built from a ranked set of requirements, which may be contradictory. The requirements are "conflict tolerant", i.e., when overruled, they continue giving "advice." This is achieved through means closely related to our weighted edges given in Section 2.4. D'Souza and Gopinath describe how to synthesize controllers in which advice from a requirement is alternately followed and ignored. The question they answer is how to synthesize a system that always follows the highest ranked advice. The approach differs from ours in the focus on contradictory specifications rather than environment failures, and in the fact that the proper action is chosen greedily, whereas we solve a global optimization problem to find the appropriate behavior.

Also D'Ippolito et al. [61, 60] consider ranked sets of requirements with incremental guarantees based on increasingly strengthened assumptions. They define a stack of specifications with strong assumptions and guarantees on top and weak assumptions and guarantees at the bottom. They propose to synthesize a controller for each rank and follow the advice of the highest ranked controller for the given environment actions. This results in graceful degradation when the environment does not behave as expected (follow advice of a lower ranked controller), and progressive enhancement when it does (follow advice of a higher ranked controller). This notion of robustness is similar to our notion of robustness in that for unexpected environment behavior the user has to specify the wanted system/controller behavior. On the other hand, the environment in our case of reactive system synthesis is only restricted by the environment assumptions whereas it is given as environment model in the case of controller synthesis.

Damm and Finkbeiner [52] consider the combination of uncontrollable envi-

ronment variables and system requirements with different priorities. A strategy is remorsefree if no other strategy satisfies a higher priority requirement for every trace of uncontrollable environment variables. They answer the question whether it is necessary to model an uncontrollable environment variable in order to find a remorse free strategy. In contrast we do not prioritize requirements, and we do not synthesize dominating strategies. Our robust systems satisfy the specification and try to satisfy it as well as possible for unexpected environment behavior.

2.6.4 Games

Closely related to the question of what should happen in reactive system synthesis if the environment does not behave as expected is the question of what should happen if a game is lost. We discuss two works that consider appropriate behavior when a game is lost. Additionally, we shortly discuss Büchi ranking games used for synthesis of prioritized requirements and lexicographic mean-payoff games used for synthesis of quantitative objectives.

Faella [71] considers the question of the appropriate behavior when a game is lost. He considers two notions, one based on dominating strategies and one based on a probability distribution over the input. In the former setting, he maximizes the set of inputs for which the game is won, and in the latter setting, the probability that the game is won. In contrast, we consider realizable specifications and propose to extend the specifications for the parts where the realizability depends on the environment behavior. It could be an interesting alternative approach for future investigation to synthesize robust systems by removing environment assumptions from the specification and then synthesizing a system that satisfies the guarantees for as many inputs as possible, while still realizing the original specification. Such an approach would have the advantage that less user input is required than in our approach. On the other hand it is not clear if the resulting system exhibits the robustness behavior one wishes for.

Chatterjee at al. [41] also consider lost games. For an unrealizable specification G, which corresponds to a lost synthesis game, they search for a maximally weak environment assumption A such that the specification $A \rightarrow G$ is realizable. In a way we also try to synthesize systems that depend as little as possible on the environment assumptions (are robust with respect to unanticipated environment behavior). In contrast to the work of Chatterjee at al. [41] we consider realizable specifications and let the user specify robustness for the case of failing assumptions.

Alur, Kanade, and Weiss [5], consider prioritized requirements and present an efficient way to synthesize the highest priority requirement using Büchi ranking games. Our robust liveness specifications described in Section 2.5 can also be considered as prioritized requirements. We try to synthesize the most robust system by synthesizing the greatest realizable robustness specification, see Section 2.5.4. Future work could consider to optimize our synthesis algorithm for robust liveness specifications by combining it with the ranking games described in [5].

Bloem et al. [23] define lexicographic mean-payoff games to synthesize reactive systems with quantitative objectives. Quantitative objectives are objectives

where the user can specify rewards for wanted behavior. For example, an arbiter should avoid unnecessary grants or long waiting times. In contrast, our work on robustness allows the user to specify errors for the environment and the system for behavior outside the wanted behavior (failing assertions and assumptions). Where we solve ratio games, the work of Bloem et al. solves lexicographic mean-payoff games. Common for both works is the general idea of adding quantitative objectives for the synthesis of reactive systems to achieve better quality for the resulting system.

2.6.5 Robustness Specifications

Peled [116] looks at the robustness of specifications. A specification formalism is robust if it can not distinguish between equivalent behavior. For example, considering stutter equivalence, a specification formalism is robust if it can not distinguish between behavior where a state (output) repeats. LTL without the X operator is robust with respect to stutter equivalence. The goal of the work is to exploit robustness properties of the specification for verification. The goal of our work is to synthesize robust systems. Synthesizing systems form robust specifications as defined in [116] would simplify the synthesis algorithm but the expressive power of the specification language is restricted and the resulting system does not necessarily satisfy any robustness criteria as we defined it.

Tabuada and Neider [133] take a different approach to robust specifications. They define robust LTL which defines a new semantic for LTL formulas. They use a many-valued semantic, assigning not only true and false to an LTL formula. For example a formula $\mathsf{G}\,p$ is true if p holds in every time step and it is false if (1) p does not hold for finitely many time steps, (2) p does not hold for infinitely many time steps, or (3) p does not hold for all time steps. The semantics of rLTL assigns different values to the three different possibilities of violating $\mathsf{G}\,p$. Intuitively a system is more robust in the first case than in the second or third case, and more robust in the second case than in the third. Looking at the semantics of an rLTL implication, the approach is similar to our definition of robustness given in Section 2.4.1, in requiring that finitely many environment errors only result in finitely many system errors. Our approach allows the user to specify the wanted robustness behavior in a lot more detail. This can be seen as an advantage or disadvantage. The additional expressive power of our way of specifying robust systems comes with additional effort on the user side, whereas rLTL formulas are LTL formulas with added semantics. The many-valued semantic is defined for all LTL operators and Tabuada and Neider show that the expressive power of rLTL is equivalent to LTL. Thus rLTL can also be seen as a short hand for LTL to specify robust systems. Our notion of measuring robustness defined in Section 2.5 is quite different in that we count the number of satisfied guarantees and do not look at how "badly" a guarantee is violated if it can not be satisfied.

2.6.6 Environment Assumptions

A good discussion on how assumptions can be handled in synthesis can be found in [24]. The paper compares different work on synthesizing systems that are

1. correct (if the assumptions are fulfilled, the guarantees are fulfilled),
2. not lazy (if possible, the guarantees are fulfilled even if the assumptions are not),
3. never give up (the guarantees are fulfilled when possible, even if they can not be fulfilled for all environment behavior), and
4. cooperate (help the environment to fulfill its assumptions).

Our robust systems are correct and not lazy, they try to satisfy the system guarantees even when the environment assumptions fail. They give up if they can not fulfill the guarantees for the worst case behavior of the environment.

The work of Bloem et al. [25] targets all four properties (correct, not lazy, never give up, and cooperate). They synthesize maximally cooperative reactive systems, meaning that the synthesized system should satisfy the guarantees and assumptions as far as possible. This is in line with our specification of robustness in the sense that the system should try to satisfy its guarantees even when the environment assumptions are violated. The advantage of cooperative reactive systems is that the user does not have to specify robustness, on the other hand the notion is more abstract. Considering Example 2 from Section 2.4, we can make two observations. First, the environment assumptions are independent of the system behavior, thus the system can not cooperate to satisfy the environment assumptions. Second, to satisfy the system guarantees, the system relies on the environment assumptions, thus the system can not satisfy its guarantees without the cooperation of the environment. Thus I claim that all systems that realize the specification of Example 2 can only reach the lowest level of cooperation. Still we show that realizing systems with different levels of robustness exist.

Eisner considers properties in CTL of the form $f = \mathsf{AG}\, g$ (g always holds) and calls a system robust if f holds in all states, not just in the reachable states. This implies that the system does not depend on environment assumptions and that it behaves well in the presence of environment failures. Eisner states that control-intensive applications are typically not robust [67].

2.6.7 Extensions

The work presented in [21] extends the work presented here by combining the two notions of robustness for GR(1) properties. Given a double cost automaton defining the error specification as defined in Section 2.4.1, a liveness robustness specification as defined in Section 2.5.1, and a ratio k, the problem of synthesizing a system that is at least k-robust can be solved with a mean payoff parity game.

The games we consider look for an optimal/winning strategy for the worst case behavior of the environment. Chatterjee et al. [42] also consider probabilistic

environments. In addition to a qualitative and a quantitative objective the user provides a distribution of the inputs and the synthesis algorithm tries to find the optimal system with respect to the average across system behaviors.

In addition to the theoretical combination of safety and liveness robustness, Bloem et al. [21] also present an implementation and a case study for the synthesis of liveness-correct and safety-robust systems. Liveness-correct and safety-robust systems are systems that realize the liveness specification and are robust with respect to an error specification. The synthesis problem can be solved with a two pair Streett game, where one Streett pair encodes the safety-robustness objective and the other Streett pair encodes the liveness correctness objective. The case study shows that the robustness requirement is quite costly with respect to the size of the resulting system and the synthesis time.

Ehlers presents another extension of our work in [65]. He extends the specifications to generalized Rabin specifications and adds a time bound for safety violations.

2.7 Conclusions

We define robustness for reactive systems in the context of LTL synthesis. The specifications we consider are of the form $A \rightarrow G$, where A is an environment assumption and G is a system guarantee. If A and G are safety specifications, a system is robust if the ratio of the number of environment errors to system errors is small. Intuitively this means that the number of errors the system makes must be proportional to the number of errors the environment makes. If A and G are liveness specifications, the number of violations can not be counted. For liveness specifications we count how many of the assumptions in A and guarantees in G are violated. A system is robust if it satisfies as many guarantees as possible for any number of satisfied assumptions. Intuitively this means that the system tries to satisfy its guarantees even if the environment does not satisfy its assumptions. Both, in the safety and the liveness case, the system behaves reasonable even when the environment behaves unexpectedly.

Work presented in this chapter shows how to verify if a given system satisfies an error specification (safety) or a robustness specification (liveness), and we give a synthesis algorithm to synthesize a robust system from an error specification (safety) or robustness specification (liveness). The development of the synthesis algorithms led to new game theoretic results. First we define ratio games, show that they have optimal positional strategies, and show how to calculate an optimal positional strategy in pseudopolynomial time. Second we develop a novel game-theoretic algorithm to solve games with GR(1) objectives.

In order to be able to apply LTL synthesis it is crucial that the synthesized systems exhibits certain quality properties like minimal size or robustness. In addition the user must not be burdened with the task to specify the full behavior in detail. Our method relieves the user from specifying the system behavior in case of environment failures and at the same time allows the user to define an error specification or robustness measure which gives the flexibility to decide

what is best suited for the system to be synthesized.

3

Security Policy Modeling for Smart Cards

This chapter discusses security policy modeling as required by Common Criteria version 3.1 [36] for certifications with Evaluation Assurance Level 6 and Evaluation Assurance Level 7. It includes the work of two publications: [16] and [84].

3.1 Problem Statement

For high security Integrated Circuit (IC)s, a security evaluation by an independent institution is of great importance to strengthen the confidence in the security of the product. Common Criteria is a widely used evaluation method for security products. In many countries, Common Criteria evaluations are required by law for certain Information Technology (IT) products such as passports or identity cards. For high Evaluation Assurance Levels, Common Criteria requires a formal model of the implemented security policies. Since formal modeling is not widely spread in industry, a Security Policy Model usually needs to be done in addition to the established design and documentation process. Additionally it requires expertise in the field of formal methods, which is not always easily available.

Not many Common Criteria certifications have reached high evaluation levels Evaluation Assurance Level 6 and Evaluation Assurance Level 7. Before 2012 only seven Common Criteria certificates were issued with an evaluation assurance level higher than 6. Figure 3.1 shows the statistics taken from the Common Criteria portal `https://www.commoncriteriaportal.org/products/stats/`. In the 'ICs, Smart Cards and Smart Card-Related Devices and Systems' category the first Evaluation Assurance Level 6 certification was reached by STMicroelectronics in 2009 in the French scheme, where they modeled a dynamic Memory Access Control Policy for the ST23 platform. Unfortunately no detailed description of

EAL	1999 - 2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	Total
EAL1	2	10	3	1	0	1	10	2	2	2	33
EAL1+	1	17	0	2	11	2	0	1	2	0	36
EAL2	4	29	5	10	2	11	2	12	17	6	98
EAL2+	7	27	14	18	15	17	32	27	36	26	219
EAL3	3	18	7	3	26	33	23	11	12	4	140
EAL3+	4	40	11	16	18	34	46	27	23	5	224
EAL4	3	35	6	9	5	6	2	7	3	0	76
EAL4+	16	149	63	74	65	68	106	69	51	17	678
EAL5	0	6	3	2	0	1	0	0	0	0	12
EAL5+	3	50	27	31	43	35	29	58	55	15	346
EAL6	0	0	0	0	0	0	0	0	0	0	0
EAL6+	0	0	0	2	3	0	4	8	6	4	27
EAL7	0	0	0	1	0	0	0	4	0	0	5
EAL7+	0	0	0	0	1	0	0	0	0	0	1
Basic	0	0	0	0	0	0	0	0	0	0	0
Medium	0	0	0	0	0	0	0	0	0	0	0
US Standard	0	0	0	0	0	0	0	0	0	0	0
None	0	0	0	0	0	0	1	9	51	33	94
Totals:	43	381	139	169	189	208	255	235	258	112	1989

Table 3.1: Certified Products by Assurance Level and Certification Date,
source: `https://www.commoncriteriaportal.org/products/stats/`

the model or the used method is publicly available.

In 2011 NXP aimed for an Evaluation Assurance Level 6 certification of the P60 Secure Smart Card Controller in the German scheme, certified by the Bundesamt für Sicherheit in der Informationstechnik. The certification is based on the Common Criteria assurance requirements [38] and the guidelines [2] and [3], but no best practices or established processes are available.

The main challenge is to develop an efficient method for security policy modeling that satisfies the Common Criteria requirements and guidelines and is suitable for hardware and software. Additionally the method should allow to leverage the advantages of a formal model to decrease time to market, decrease costs in late bug fixes and increase product quality and security. To maximize the benefits of a formal model the modeling has to be integrated into the development process of the product. To allow integration it is necessary to have a comprehensive and intuitive modeling method. Such a method must be supported by user-friendly tooling that is quickly accepted by the engineers using it. Currently it is often hard to convince engineers to use formal methods due to bad usability of the relevant tools and because considerable effort needs to be invested on first use and before first results are visible. For most formal methods tools, an expert is needed to use it to its full extend.

In their review of industrial practice in formal methods Bicarregui et al. state "Significant challenges remain in providing usable tools that can be integrated into established development processes; in education and training; in taking formal methods from first use to second use, and in gathering and evidence to support informed selection of methods and tools." [17].

3.2 Contribution

In the first part of this chapter (Section 3.4) we show how a formal security policy model based on temporal logic and model checking can be developed for the real world evaluation of a Security IC. We argue that temporal logic and model checking is suitable for the formal requirements of a Common Criteria Evaluation Assurance Level 6 evaluation, see Section 3.4.2. Model checking is an efficient method because modeling the functional specification and formalizing the security requirements can be done by anybody with moderate knowledge of formal methods. Additionally, proofs (or refutations) are generated automatically which is a big advantage compared to theorem proving where proofs often need to be guided by the user. Previous work proposed to use theorem provers for security policy modeling, see Section 3.6.1. Our work is the first to provide a Security Policy Model for an Evaluation Assurance Level 6 certification based on model checking. This work led to the first Evaluation Assurance Level 6 certificate for an IC in the German scheme, see `https://www.commoncriteriaportal.org/products/`.

In addition to describing the used method, we provide descriptions of two security policy models. First we describe the Security Policy Model of the access control policy of the NXP Secure Smart Card Controller P60x144/080PVA. The model was part of the evaluation evidence of the first Evaluation Assurance

Level 6 certificate for an IC in the German scheme. Second we describe the model of the firewall access control policy of the Java Card specification. The second model revealed inconsistencies between the Java Card specification and the Protection Profile. This shows that errors in product requirements which are not detected for several years can easily be uncovered with our Security Policy Modeling method. Following our request these inconsistencies in the Java Card Protection Profile will be resolved in the next version of the Protection Profile.

In the second part of this chapter (Section 3.5) we show how Security Policy Modeling can be integrated into a design process of a smart card in an industrial context by formally verifying Unified Modeling Language (UML) statecharts. We argue that modeling with UML statechart diagrams is generally feasible for non formal methods experts and the diagrams can be used both as part of the documentation and in formal verification for security policy modeling. Additionally the UML statecharts can be used for test case generation closing the gap between the formally verified specification and the implementation (test case generation is part of related work see Section 3.6.3, building on work described here).

Integrating Security Policy Modeling into the design process of a product eliminates inconsistencies in the documentation and removes ambiguities in the specification of modeled parts. Security holes in the specification can be detected and fixed before the implementation is started. This increases the quality of the documentation and the security of the product.

We present possible tooling and an example Security Policy Model to illustrate the process. Our goal is to move formal verification from an isolated task done by an expert in formal methods to the center of the design process by proposing easy-to-use tooling support. The work presented here is a first step into this direction.

3.3 Preliminaries

This section will give a general introduction on smart cards, Common Criteria, model checking, and UML statecharts.

3.3.1 Smart Cards

Smart cards are small devices usually in a card form factor that are present in our daily lives. We use them

- for payment in the form of credit/debit cards,
- for loyalty programs like frequent flyer,
- for ticketing to access public transportation for example with a prepaid card,
- for accessing buildings or rooms as for example often used in hotels in form of an access card,

- for identification in the form of an electronic passport or identity card,
- for secure information storage in the form of health/insurance cards.

The number of devices in the field grows rapidly, for example the number of payment cards has grown from 880 million devices shipped in 2010 to 2050 million devices shipped in 2014 according to Eurosmart (`http://www.eurosmart.com/publications/market-overview`). Not only the number of cards grows but also the complexity of the devices, moving from dedicated hardware for one application to flexible programmable hardware supporting multiple applications.

A general architecture of a modern smart card is given in Figure 3.1. It consists of an underlying security IC platform, some native applications, a Java Card OS, and applets. The four components will be described in more detail in the following.

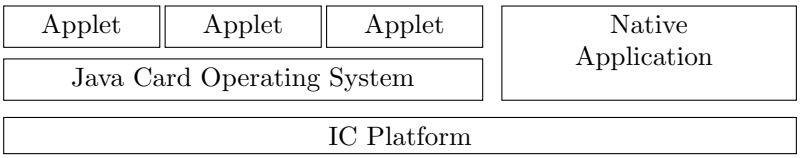


Figure 3.1: Architecture of a smart card

Security IC Platform

A typical security IC is a micro controller with

- a Central Processing Unit (CPU) that executes the operations,
- an Memory Management Unit (MMU) that manages and controls the access to the memories,
- the memories, with Read Only Memory (ROM) to store software, Random Access Memory (RAM) to store data and code during execution, Electrically Erasable Programmable Read Only Memory (EEPROM) and/or flash to store data and software,
- contact or contactless communication interfaces to communicate with a reader,
- a random number generator to support key generation and other cryptographic operations,
- cryptographic coprocessors to speed up cryptographic operations such as encryption, decryption, signing, signature verification etc., and
- security circuitry like sensors to detect fault attacks.

Figure 3.2 shows the typical components of a security IC. The security circuitry is not part of the figure, it is spread over the whole circuit. A more detailed description can be found in [32].

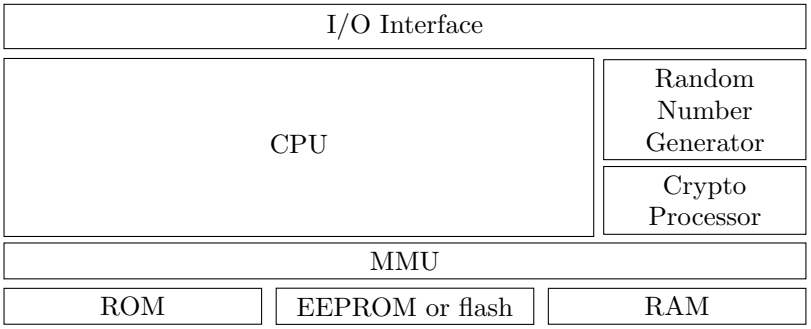


Figure 3.2: Components of a Typical Security IC

Native Application

A native application is software that runs directly on the security IC platform. Its functionality can be accessed by the end user directly. Examples are applications of the MIFARE family such as MIFARE Plus and MIFARE DESFire which are used for electronic fare collection, stored value card systems, access control systems, and loyalty cards.

Java Card OS

A Java Card OS is an operating system that runs directly on the security IC platform. It provides a (restricted) Java interface to the applet developer. The definition of the Java Card Application Programming Interface (API) can be found in [113]. The standardized interface is one of the big advantages of the Java Card OS. It allows to port applets from one Java Card to another.

A Java Card OS consists of the Java Card Virtual Machine (JCVM) defined in [115] and the Java Card Runtime Environment (JCRE) defined in [114]. Additionally most Java Card OS implement card management as defined by GlobalPlatform [82].

Like the classic Java Virtual Machine, the Java Card Virtual Machine executes bytecode. It only supports a subset of the classic bytecodes and optimizes for space in order to allow execution on a small platform with restricted resources.

The JCRE supports logical channels, applet selection, transient objects, applet isolation, and object sharing among other management functionality. The most important security feature defined in the JCRE is the applet firewall. The firewall ensures that data of one applet can not leak to another applet.

Applet

An applet is a Java Card application that runs on the Java Card OS. Typical examples are banking applets as defined in [70], or e-government applets as defined by the International Civil Aviation Organization (ICAO).

3.3.2 Common Criteria

Not only the complexity of smart cards grows but also the demand on security, applications become more security critical and hacking techniques more sophisticated. Especially for banking and e-government a high assurance of the implemented security is essential.

Common Criteria is a security certification scheme that allows to compare the security level of different smart card implementations and gives the user confidence in the certified product. The certificate is issued by an independent third party. The certification authority is usually a governmental organization such as the Bundesamt für Sicherheit in der Informationstechnik.

There are three parties involved in a certification, the developer, the evaluator, and the certification authority. The developer (for example NXP Semiconductors) provides the Target of Evaluation with all the required documentation. The evaluator examines the Target of Evaluation and the documentation according to the Common Criteria methodology [39] and writes a report for the certification authority. After reviewing the evaluation report the certification authority issues the certificate. A lengthy introduction to Common Criteria can be found in [35].

The main idea of the Common Criteria certification scheme is to have comparable and standardized security requirements and assurance requirements. The Security Target is the central document of the certification, it states the security requirements that are evaluated ('what' is evaluated) and the assurance requirements that are applied in the evaluation ('how' is it evaluated). Both security requirements and assurance requirements are taken from a standardized catalog of requirements, [37] and [38] respectively.

Below we list the security requirement classes that are available in [37]. The list gives an overview of the security concerns considered by Common Criteria. Not all of them are relevant for this work. For example, for a security IC the main threads are manipulation or disclosure of data stored on the IC and manipulation of the functionality of the IC. These threads lead to different security requirements. For example a security IC needs to provide physical protection against tampering (for example fault attacks) to counter physical manipulation. This requirement is part of the security requirement class for the protection of the Target of Evaluation Security Functionality. Another security requirement for a security IC could be the separation of memory defined by an access control policy to counter disclosure and manipulation of data. Access control policies are part of the security requirement class for user data protection. More details on threads and security requirements for security ICs can be found in the Protection Profile [33] or the Security Target [110] which we use in our case study in Section 3.4.3. For a Java Card OS the main threads are similar to the threads for a security IC. The security requirements differ a bit but they belong to the same security requirement classes. A description can be found in the Protection Profile [122] which we used in our case study in Section 3.4.4.

The following security requirement classes are available in [37]:

- security audit (recognizing, recording, storing, and analyzing information related to security relevant activities),

- communication (assuring the identity of a party participating in a data exchange),
- cryptographic support (key management and cryptographic operation),
- user data protection (within a Target of Evaluation, during import, export, and storage),
- identification and authentication (establish and verify a claimed user identity),
- security management (management of security attributes, Target of Evaluation Security Functionality data and functions),
- privacy (user protection against discovery and misuse of identity by other users),
- protection of the Target of Evaluation Security Functionality (integrity and management of the mechanisms that constitute the Target of Evaluation Security Functionality and integrity of the Target of Evaluation Security Functionality data),
- resource utilization (availability of required resources),
- Target of Evaluation access (controlling the establishment of a user's session), and
- trusted path/channels (trusted communication path between users and the Target of Evaluation Security Functionality and a trusted communication channel between the Target of Evaluation Security Functionality and other trusted IT products).

Every security requirement class defines a list of Security Functional Requirements. The requirements in the Security Target are taken from these lists. Note that Security Functional Requirements might have dependencies, which means that an Security Functional Requirement might require to include other Security Functional Requirements, for example to define a policy (see Section 3.4.4 for an example). The main policies are the access control policy and information flow control policy, they are defined in the assurance class for user data protection.

Below we list the assurance requirement classes that are available in [38]. Common Criteria defines requirements for the whole development cycle. Documentation and evidence needs to be provided by the developer to the evaluator to show that the development follows the required processes from design to test. For example life cycle support requires a configuration management system to be used in the development. For the work presented here we only consider the assurance requirements for the Security Policy Model as listed in Section 3.4.2.

The following assurance requirement classes are available in [38], the text copied from [38] is given in *italics*:

- Development (ADV) - *The requirements of the Development class provide information about the Target of Evaluation. The knowledge obtained by this information is used as the basis for conducting vulnerability analysis and testing upon the Target of Evaluation, as described in the AVA (vulnerability assessment) and ATE (tests) classes.*

- Guidance Documents (AGD) - *The guidance documents class provides the requirements for guidance documentation for all user roles. For the secure preparation and operation of the Target of Evaluation it is necessary to describe all relevant aspects for the secure handling of the Target of Evaluation. The class also addresses the possibility of unintended incorrect configuration or handling of the Target of Evaluation.*

- Life-cycle support (ALC) - *Life-cycle support is an aspect of establishing discipline and control in the processes of refinement of the Target of Evaluation during its development and maintenance. Confidence in the correspondence between the Target of Evaluation security requirements and the Target of Evaluation is greater if security analysis and the production of the evidence are done on a regular basis as an integral part of the development and maintenance activities.*

- Security Target evaluation (ASE) - *Evaluating an Security Target is required to demonstrate that the Security Target is sound and internally consistent, and, if the Security Target is based on one or more Protection Profiles or packages, that the Security Target is a correct instantiation of these Protection Profiles and packages. These properties are necessary for the Security Target to be suitable for use as the basis for a Target of Evaluation evaluation.*

- Tests (ATE) - *Testing provides assurance that the Target of Evaluation Security Functionality behaves as described (in the functional specification, Target of Evaluation design, and implementation representation).*

- Vulnerability assessment (AVA) - *The vulnerability assessment class addresses the possibility of exploitable vulnerabilities introduced in the development or the operation of the Target of Evaluation.*

The Security Policy Model is part of the assurance class development. Figure 3.3 illustrates the relationships between different parts of the assurance class development. As can be seen in the figure, the development class follows the waterfall model. First the functional requirements stated in the Security Target are refined into the functional specification which describes the Target of Evaluation Security Functionality Interfaces. This first refinement step is part of the Functional Specification assurance family. In the next step the functional specification is refined into the design description which describes the modules and internal structure of the design. This second refinement step is part of the Target of Evaluation Design assurance family. The last refinement step maps the design description to the implementation representation (IMP). Together the

three refinement steps ensure that the requirements stated in the Security Target are implemented in the Target of Evaluation.

The policy model formalizes the first refinement step from functional requirements of the Security Target to the functional specification. The Security Policy Model requires a mapping both to the functional requirements and the functional specification. A list of the requirements for the Security Policy Model is given in Section 3.4.2.

Figure 3.3: Documentation Refinement in the Assurance Class Development

An Security Policy Model is not required for all Evaluation Assurance Levels. Every Evaluation Assurance Level requires a certain list of assurance components. Figure 3.4 shows the required assurance components for each Evaluation Assurance Level. For each assurance family, assurance component '1' is the lowest assurance component, meaning that it includes the minimum assurance requirements for this assurance family. With increasing assurance component numbers more assurance requirements are added for the respective assurance family. For example Functional Specification.4 requires a complete functional specification, Functional Specification.5 a complete semi-formal functional specification, and Functional Specification.6 a complete semi-formal functional specification with additional formal specification. With increasing Evaluation Assurance Level also the necessary assurance component numbers increase. For example Evaluation Assurance Level 5 requires Functional Specification.5, no Security Policy Model, and Target of Evaluation Design.4 whereas Evaluation Assurance Level 6 requires Functional Specification.5, Security Policy Model.1, and Target of Evaluation Design.5. Note that the Functional Specification component does not change from Evaluation Assurance Level 5 to Evaluation Assurance Level 6 but Evaluation Assurance Level 6 adds the Security Policy Model.1 component and moves from Target of Evaluation Design.4 to Target of Evaluation Design.5. Note that the Security Policy Model has only one component which is only required for Evaluation Assurance Level 6 and Evaluation Assurance Level 7. The requirements for all components are given in [38]. A methodology description for the evaluation is given in [39]. Additional guidance for high Evaluation Assurance Level certifications and formal methods is given in [2] and [3].

For certain product families a template for the Security Target exists, a so called Protection Profile. The Protection Profile defines the minimum functional requirements and assurance requirements for a product family. The intention is to increase comparability. Examples are [32] for Security IC Platforms, and [122] for Java Cards. Although the minimum required assurance level for security ICs is Evaluation Assurance Level 4 augmented with AVA_VAN.5 and ALC_DVS.2, the established standard in the industry is Evaluation Assurance Level 5 augmented with AVA_VAN.5 and ALC_DVS.2. Only few products have reached Evaluation Assurance Level 6 and higher.

Assurance class	Assurance Family	Assurance Components by Evaluation Assurance Level						
		EAL1	EAL2	EAL3	EAL4	EAL5	EAL6	EAL7
Development	ADV_ARC		**1**	1	1	1	1	1
	ADV_FSP	**1**	**2**	**3**	**4**	**5**	5	**6**
	ADV_IMP				**1**	1	**2**	2
	ADV_INT					**2**	**3**	3
	ADV_SPM						**1**	1
	ADV_TDS		**1**	**2**	**3**	**4**	**5**	**6**
Guidance documents	AGD_OPE	**1**	1	1	1	1	1	1
	AGD_PRE	**1**	1	1	1	1	1	1
Life-cycle support	ALC_CMC	**1**	**2**	**3**	**4**	4	**5**	5
	ALC_CMS	**1**	**2**	**3**	**4**	**5**	5	5
	ALC_DEL		**1**	1	1	1	1	1
	ALC_DVS			**1**	1	1	**2**	2
	ALC_FLR							
	ALC_LCD			**1**	1	1	1	**2**
	ALC_TAT				**1**	**2**	**3**	3
Security Target evaluation	ASE_CCL	**1**	1	1	1	1	1	1
	ASE_ECD	**1**	1	1	1	1	1	1
	ASE_INT	**1**	1	1	1	1	1	1
	ASE_OBJ	**1**	**2**	2	2	2	2	2
	ASE_REQ	**1**	**2**	2	2	2	2	2
	ASE_SPD		**1**	1	1	1	1	1
	ASE_TSS	**1**	1	1	1	1	1	1
Tests	ATE_COV		**1**	**2**	2	2	**3**	3
	ATE_DPT			**1**	1	**3**	3	**4**
	ATE_FUN		**1**	1	1	1	**2**	2
	ATE_IND	**1**	**2**	2	2	2	2	**3**
Vulnerability assessment	AVA_VAN	**1**	**2**	2	**3**	**4**	**5**	5

Figure 3.4: Evaluation Assurance Summary, source [38]

3.3.3 Model Checking

Model checking is a formal verification technique that allows to fully automatic check if a finite state system fulfills certain properties. The results are based on a systematic exhaustive inspection of all states of the mathematical model, the so-called state space [50]. The application of model checking consists of three tasks:

- *Modeling* is the creation of a formal model of the system as a Finite State Machine (FSM) (for example Moore Machine see Section 2.3.1) with the input language of a model checker.
- *Formalizing* is the derivation of properties in terms of temporal logic formulas from functional or security requirements.
- *Verification* is the task of running the model checker which carries out the check whether the properties are valid on the given model. If the model satisfies a property the model checker outputs `true`, otherwise a counter example is presented to the user. The counter example shows how the model violates the given property.

Model checking is typically applied to sequential hardware circuits or communication protocols [10]. In contrast we apply it to abstract specifications of smart cards, in particular to specifications of hardware and software components.

We use the model checker NuSMV [48]. NuSMV has a proprietary input language to describe the FSM model. The following items need to be defined in NuSMV to describe a FSM:

- Variables - The variables define the state space of the model. They can be input variables which are set non-deterministically by the environment, or internal/output variables which are set deterministically as defined by the state transitions. Optionally the values of input variables can be restricted using invariants (INVAR).
- Initial state - The initial state is an assignment to all variables in the first time step.
- Transitions - The state transitions define the internal/output variable assignment in the next time step depending on the current assignment of all variables.

The diameter of a NuSMV model is the longest run such that no state appears twice in the run. For more details we refer to the NuSMV user manual [73].

We use Linear Temporal Logic (LTL) (LTLSPEC), Computation Tree Logic (CTL) (CTLSPEC), and invariant specifications (INVARSPEC) to formalize the requirements. Refer to Section 2.3.4 for a description of temporal logic.

For the work described in Section 3.5 we use the statechart modeling and formal verification feature of COSIDE. COSIDE provides a front end to NuSMV. The input to COSIDE is an UML statechart diagram as described in Section

3.3.4 and a specification file with LTL properties, CTL properties, and invariants. It translates the UML statechart diagrams into NuSMV models, and runs the model checker NuSMV on the models and the specification file. If a counter example is generated by NuSMV, COSIDE provides the possibility to run it on the UML statechart diagram.

When reading the size of our models in the following sections, note that there is not a one to one mapping from the states of a UML statechart to the states of a NuSMV model. The states in a UML statechart are a graphical representation of a subset of internal variables of an FSM, whereas the state space in a NuSMV model is the set of all possible variable assignments.

3.3.4 Unified Modeling Language Statechart Diagrams

The UML defines a standard for the modeling of statecharts [112]. It offers a definition of the graphical notation, the abstract syntax as well as an informal description of the semantics. Statecharts describe the dynamic behavior of a system as a state transition system, i.e., the relation between inputs and outputs based on the internal state.

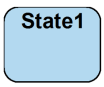
UML models are used in model driven engineering and model based testing. We use them for formal verification of specifications because they are a widely known and common tool for modeling specifications.

We use the design environment COSIDE [76] for statechart modeling and formal verification. For a full description of the UML modeling features supported by COSIDE refer to [75]. Here we only describe the features we use in our example in Section 3.5.3.

- Input, internal, and output variables:

  Input variables are set non-deterministically by the environment, internal and output variables are determined by the statechart. We use Boolean and integer variables with a given range.

- Simple states:

  Simple states can include entry or exit activities. The activity assigns values to internal and output variables if the state is entered, respectively exited. A state is entered when a transition leading to that state is taken, a state is exited when a transition leaving that state is taken.

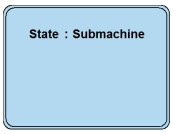
- The initial state:

  Every statechart (also substatechart) has to have exactly one initial state. The initial state has exactly one outgoing transition. The initial transition

of the statechart has to assign initial values to all internal and output variables. This is done with the activity element of the transition, see the transition description below. The initial transition has no trigger and no guard. Since there is only one transition it has the highest priority (1).

- Submachine states:



A submachine state is a state that includes a substatechart. This is a graphical element that helps to structure the statechart by allowing to have a submachine state representing a whole statechart at an upper level. The substatechart has an initial state. If a transition leads to the submachine state at the upper level then the substatechart starts execution in the initial state. The submachine state can also have an outgoing transition. This is a transition that is taken from any state of the substatechart. If the priority of the outgoing transition is 0 then it has higher priority than all other transitions in the substatechart.

- Transitions:

Transitions connect the current state with the next state. Transitions have a priority, a trigger, a guard, and an activity:

  - The priority is an integer that gives the order in which the transitions are evaluated. This is a not a standard UML feature. It helps to simplify the transitions since it allows to specify 'else' transitions, for example if a transition with priority one can only be taken if "var" is true and transition with priority two has no guard, then the second transition can be taken if "var" is false. Additionally the priorities remove non-determinism from the transitions. If for two transitions the guard evaluates to true then the one with higher priority is taken.
  - The trigger is an input variable which triggers the transition, meaning that the transition is taken if the input variable is true.
  - The guard is a propositional formula consisting of input and internal variables. If the propositional formula of the guard evaluates to true the transition is taken.
  - The activity assigns values to internal and output variables if the transition is taken, for example if the transition is the transition with the highest priority where the trigger is true and the guard evaluates to true.

If a transition is taken the statechart moves to the next state and continues evaluation there.

The statechart is executed with an abstract notion of time in the form of steps. In each step the statechart is in exactly one active simple state. The transitions between states and the execution of activities take no time. If no transition is taken, there is an implicit self-loop for simple states.

3.4 Modeling Smart Cards

This chapter shows how model checking can be used for formal security policy modeling of a smart card. First we give a description of the general approach and show how this satisfies the Common Criteria requirements for the Security Policy Model. Then we describe two case studies. The first case study describes a security policy formalization of a security IC that was developed for a concrete product evaluation. The second case study describes a security policy formalization of a Java Card System. For an introduction to smart cards refer to Section 3.3.1.

3.4.1 Modeling Approach

This section describes the general approach we use for security policy modeling for high assurance level Common Criteria certifications. Compared to other approaches which often use theorem proving (see Section 3.6.1), our approach builds on model checking.

The main reason we decided to use model checking for security policy modeling is because model checking does not require manual proofs or manual assistance in the proof. A model checker proves that the model satisfies the specification without any interaction with the user, and in case a property does not hold the model checker returns a counter example. The counter example helps in debugging the model or localizing the error in the specification. Additionally a model (FSM) is easier to understand and review by engineers without formal methods background than complicated mathematical definitions and proofs as used in theorem proving. Effectiveness and usability were the main factors in our decision.

A model checker takes (1) a model and (2) a set of properties as input and proves that the model satisfies the properties or gives a counter example. Refer to Section 3.3.3 for an introduction to model checking.

We propose the following approach.

1. The model is derived from the functional specification. The model formalizes the security-critical parts of the Functional Specification, it describes the behavior of the product at a high level of abstraction.

2. The properties we want to prove with the model checker are derived from the Security Functional Requirements of the Security Target.

Figure 3.5 illustrates the process of model checking functional specifications. The requirements on the left hand side of the figure are the Security Functional Requirements from the Security Target. On the right hand side of the figure the Functional Specification is modeled.

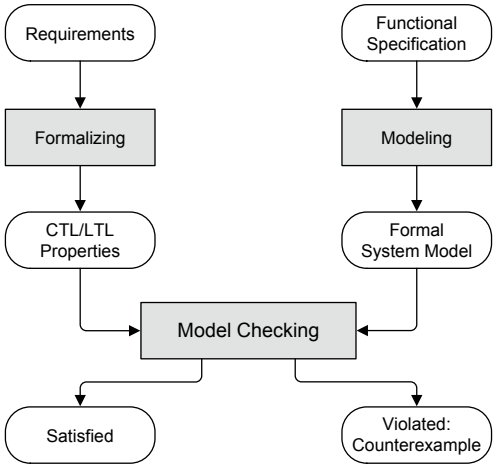


Figure 3.5: Process of Model Checking for Functional Specifications

If the model checker returns `true` we have a formal proof that the model of the Functional Specification satisfies the formalization of the Security Functional Requirements of the Security Target. If a counter example is returned, it can be evaluated if the model is wrong or the underlying specification itself is ambiguous and/or erroneous.

The two rather high abstraction levels are chosen for formalizing and modeling to satisfy the Common Criteria requirements for Security Policy Modeling. Section 3.4.2 gives a detailed explanation on how this modeling approach satisfies Common Criteria.

3.4.2 Common Criteria Requirements

This section shows how the modeling approach defined in 3.4.1 satisfies the Common Criteria assurance requirements for the Security Policy Model.

For the Security Policy Model, Common Criteria requires a formal model which shows that the Functional Specification is (i) consistent, (ii) correct and, (iii) complete with respect to the Security Policies. The Security Policy Model formalizes the refinement step from the Security Target to the Functional Specification. See Section 3.3.2 for an introduction on Common Criteria.

In the following we list the Security Policy Model requirements given in [38] and for each requirement we give a rationale how we satisfy the requirement with our approach.

The following developer action elements are part of ADV_SPM.1:

- *ADV_SPM.1.1D The developer shall provide a formal security policy model for the [assignment: list of policies that are formally modeled].*

  Rationale: This developer action element requires to list the policies that are modeled. The scope of the model is determined by the Security Target and the state of the art of the chosen modeling method. At least one policy must be modeled. Model checking is best suited for modeling access control policies because usually access control policies can easily be described by a state FSM. Examples of access control policies are

  - the access control policy of the MMU which describes the access to memories and Special Function Register for security ICs, see [110] and Section 3.4.3,
  - the MIFARE Plus access control policy that describes the access to blocks and cryptographic keys, see [111],
  - the MIFARE DESFire access control policy that describes the access to data, applications, data files, values and cryptographic keys, see [111],
  - the firewall access control policy which is the main access control policy for Java Cards, see [122] and Section 3.4.4.

- *ADV_SPM.1.2D For each policy covered by the formal security policy model, the model shall identify the relevant portions of the statement of Security Functional Requirements that make up that policy.*

  Rationale: A policy is a list of Security Functional Requirements. The list of Security Functional Requirements that define a policy is given by the Security Target. All Security Functional Requirements of the Security Target that explicitly mention the policy plus all Security Functional Requirements that they depend on are part of the policy. Not all parts of an Security Functional Requirement can always be modeled. We map the Security Functional Requirements to the formal properties and add textual explanation to identify the relevant portions of the statement of Security Functional Requirements that are modeled.

- *ADV_SPM.1.3D The developer shall provide a formal proof of correspondence between the model and any formal functional specification.*

  Rationale: For Evaluation Assurance Level 6 only ADV_FSP.5 is required (see Figure 3.4) which does not require a formal functional specification. Thus the item is vacuously satisfied. For Evaluation Assurance Level 7, ADV_FSP.6 requires a formal functional specification. In this case the model (FSM) itself could be used as the formal functional specification, especially when using UML statecharts (see Section 3.5). Thus also in this case the item would be vacuously satisfied because the model is the formal functional specification.

- *ADV_SPM.1.4D The developer shall provide a demonstration of correspondence between the model and the functional specification.*

  Rationale: We link the model to the functional specification with a table that maps the variables and transitions of the model to the semi-formal functional specification. In case Evaluation Assurance Level 7 is targeted and a formal functional specification is given the linking can be omitted since the model is the formal functional specification.

The following content and presentation of evidence elements are part of ADV_SPM.1:

- *ADV_SPM.1.1C The model shall be in a formal style, supported by explanatory text as required, and identify the security policies of the Target of Evaluation Security Functionality that are modeled.*

  Rationale: The model is given in NuSMV and temporal logic (LTL and CTL), which are formal languages. Explanatory text is given as shown in Section 3.4.3 and Section 3.4.4. The security policies are identified as described above.

- *ADV_SPM.1.2C For all policies that are modeled, the model shall define security for the Target of Evaluation and provide a formal proof that the Target of Evaluation cannot reach a state that is not secure.*

  Rationale: Security is defined for the policy through the Security Functional Requirements. The model checker proves that the model satisfies the Security Functional Requirements thus showing that the model of the security policy of the Target of Evaluation can not reach a state that is not secure.

- *ADV_SPM.1.3C The correspondence between the model and the functional specification shall be at the correct level of formality.*

  Rationale: For Evaluation Assurance Level 6 the correspondence between the model and the functional specification needs to be semi-formal. We cover this with a table that maps the variables of the model to the semi-formal description of the Functional Specification (see Section 3.4.3 and 3.4.4 for examples). For Evaluation Assurance Level 7 such a mapping would not be necessary if we would use the model itself as formal description in the Functional Specification.

- *ADV_SPM.1.4C The correspondence shall show that the functional specification is consistent and complete with respect to the model.*

  Rationale: We give a mapping for every variable used in the model and all transitions in the model to the Functional Specification. If every variable and every transition can be mapped, the Functional Specification is consistent and complete with respect to the model.

This completes the list of developer action elements and content and presentation of evidence elements of ADV_SPM.1, but the Application Notes and Interpretation of the Scheme (AIS) 34 [2] gives further requirements that need to be considered in the evaluation. The AIS 34 requests a formal proof of internal consistency of the Target of Evaluation Security Policy Model. Internal consistency is the absence of contradictions within the Target of Evaluation Security Policy Model. In our approach the formal proof of internal consistency is given implicitly. If the model would not be internally consistent the model checker NuSMV would not accept the model and generate proofs. Since the model checker accepts the model and generates proofs, the model is internally consistent. Furthermore we can define additional properties to check if the model behaves as expected. The property of internal consistency is more relevant for theorem proving. For theorem proving contradictions within the Target of Evaluation Security Policy Model can easily lead to vacuously true properties.

Note that the AIS 34 [2] uses the terms security principles, characteristics, properties and features. We interpret the security principles to be the Security Functional Requirements stated in the Security Target, and the characteristics to be the behavior of the Target of Evaluation Security Functionality Interfaces described in the Functional Specification. The formal counterparts are the security properties written in LTL/CTL, and the features given as FSM respectively. With this interpretation all evaluator action elements stated in AIS can be fulfilled.

While developed for a concrete product evaluation of a security IC, our approach for security policy modeling is suitable for generalization to other products and product types such as native applications, Java Card OSs, and applets.

The approach was successfully applied for Evaluation Assurance Level 6 certifications, we argue that it can also be efficiently applied for Evaluation Assurance Level 7 certifications. Evaluation Assurance Level 7 has the same Security Policy Model component as Evaluation Assurance Level 6, the difference lies in other components that are related to the Security Policy Model. With respect to formal modeling Evaluation Assurance Level 7 requires a formal functional specification (ADV_FSP.6) and a formal subsystem description (ADV_TDS.6). Both should be easy to provide (at least in parts) as FSM using for example UML statecharts as described in 3.5.

3.4.3 Case Study - Security IC

This section describes the Security Policy Model of the NXP Secure Smart Card Controller P60x144/080PVA. The Security Target [110] of the product is compliant to the Security IC Platform Protection Profile [33]. This Protection Profile defines a number of security policies, only the Access Control Policy is shown here.

In the next subsection a short description of the formal model (the formalization of the functional specification) is given. The following subsection describes the formal properties (the formalization of the requirements). We conclude with a discussion of the results.

Formal Model

This section describes parts of the model of the access control policy of the NXP Secure Smart Card Controller P60x144/080PVA. The model is based on *Modes*. Specific access rights to IC components are associated with every mode, like access to certain memory areas or coprocessors. The modes are modeled by a state variable "CPU". The range of this variable is shown in Table 3.2.

System State	Description
SystemM	Mode for execution of application programs. In this mode, all resources available to application programs are accessible.
UserM	Restrictive mode for execution of application programs. Accessibility of resources is configurable in System Mode. Memory access is moderated by the MMU.
FirmwareM	Mode for emulating other Security Smart Card products. The memory used in this mode is completely separated from the memory available in System Mode and User Mode.

Table 3.2: Target of Evaluation Modes Description (excerpt)

In all CPU modes, the IC controls access to memories through the MMU. The MMU translates virtual to physical addresses. Access is controlled in two ways. First, the firmware firewall splits the memory into two parts. One part is available in Firmware Mode. The other is available in System Mode and User Mode. Second, memory can be segmented into smaller areas and access rights (readable, writable or executable) can be defined for these segments in the MMU Table.

The MMU Table stores memory access rights. Note that the MMU Table itself is stored in memory. Therefore it is possible to store an MMU Table in memory writable in User Mode. In this case a User Mode process may manipulate the MMU Table, possibly circumventing restrictions.

Modeling Memory Segmentation: MMU Tables - In order to abstract from the implementation details and the complexity of multiple tables and processes at arbitrary memory addresses, the following model is used: We explicitly model two user processes (process 0 and process 1) and one memory segment only. We also do not use explicit memory addresses, but Boolean variables ("mmuTableInSeg[0]", and "mmuTableInSeg[1]") indicating that the process' MMU table is in the modeled segment or somewhere else. Also, details of the MMU table data structure are not modeled. Only the access rights are given by the variables "MMUtable[0]" for process 0 and "MMUtable[1]" for process 1, which can be subsets of {r,w,x}. The values represent read, write, and execute access, respectively.

Modeling Memory Partitioning: Firmware Firewall - The mechanism separating Firmware Mode from System and User Mode is called "Firmware Firewall". The Firmware Firewall is modeled similar to the MMU. A state variable "FMcanAccessSegment" indicates if Firmware Mode processes have access to the

modeled memory segment.

Modeling Operations - State transitions are triggered by *operations*. Operations can be triggered by software running on the IC (for example memory write operations), external events (for example an attacker manipulating the IC), or internal events. The operations used in the formal properties in the next section are shown in Table 3.3.

Operation	Description
OpProc0SegmentWrite, OpProc1SegmentWrite	User Mode process 0/1 writes to modeled memory segment.
OpProc0SegmentAccess, OpProc1SegmentAccess	User Mode process 0/1 accesses modeled memory segment.
OpSMmemoryAccess	System Mode process accesses modeled memory segment.

Table 3.3: Operations (excerpt)

Formal Properties

This section shows an excerpt of the formalizations of the Security Functional Requirements defining the Access Control Policy for the memory given in [110]. The Security Target states: "The Access Control Policy comprises the following Security Functional Requirements: FDP_ACC.1[MEM], FDP_ACC.1[SFR], FDP_ACF.1[MEM], FDP_ACF.1[SFR] with the associated dependencies. Further the secure state as required by FDP_FLS.1 is included in the security policy model. In addition parts of the life cycle control as required by FMT_LIM.2 are part of the model."

The two main Security Functional Requirements FDP_ACC.1[MEM] and FDP_ACF.1[MEM] with the associated dependencies define the Access Control Policy for the memory. Parts of its formalization is given below. For the certification in the German scheme all policies need to be modeled. For any parts of a policy that can not be modeled an argument must be given by the developer. Below, all properties are given in the NuSMV specification language [73]. We only show formulas for process 0, for process 1 the formulas are similar.

All User Mode memory accesses are moderated by the MMU. In User Mode, memory access is possible only when the MMU Table allows it ($MMUtable[0] \neq none$). Note that here we do not distinguish between read, write, and execute accesses to simplify the model.

```
INVARSPEC
  ((CPU = UserM) & OpProc0SegmentAccess) ->
      (MMUtable[0] != none)
```

In the following formulas, the property that memory content does not change is represented by the statement

```
mmuTableInSeg[0] & (MMUtable[0] = next(MMUtable[0])).
```

While this statement explicitly talks about changes in the MMU Table of process 0, it can be interpreted as any change in the memory segments accessible by process 0.

When the CPU is in User Mode and the MMU Table is in none of the memory segments where the User Mode has access then the MMU Table of a process does not change. Note that we do not model changing an MMU Table in System Mode, because the Access Control Policy requires isolation of User Mode processes from each other only.

```
INVARSPEC
  ((CPU = UserM) & (next(CPU) = UserM) &
  (!mmuTableInSeg[0]) & (!mmuTableInSeg[1])) ->
       ((MMUtable[0] = next(MMUtable[0])) &
       (MMUtable[1] = next(MMUtable[1])))
```

In User Mode, even if the MMU Table is in a segment where the User Mode has access, the MMU Table of a process does not change unless it is writable by one of the user processes.

```
INVARSPEC
  ((CPU = UserM) & (next(CPU) = UserM) &
         mmuTableInSeg[0] &
  !(MMUtable[0] in {w, rw, wx, rwx}) &
  !(MMUtable[1] in {w, rw, wx, rwx})) ->
       (MMUtable[0] = next(MMUtable[0]))
```

User Mode Processes are isolated from each other, for example one process cannot write memory assigned to another process unless explicitly permitted.

```
INVARSPEC
  ((CPU = UserM) & (next(CPU) = UserM) & mmuTableInSeg[0]
  & !(MMUtable[1] in {w, rw, wx, rwx})) ->
       ((MMUtable[0] = next(MMUtable[0])) |
       OpProc0SegmentWrite)
```

The second part of the security properties of the Access Control Policy define the separation between memory assigned to Firmware Mode and memory assigned to the other modes.

When a memory segment is accessible in User Mode, it is not accessible in Firmware Mode.

```
INVARSPEC
  ((OpProc0SegmentAccess) | (OpProc1SegmentAccess)) ->
       !FMcanAccessSegment
```

When a memory segment is accessible in System Mode, it is not accessible in Firmware Mode.

```
INVARSPEC
  OpSMmemoryAccess -> !FMcanAccessSegment
```

When a memory segment is accessible in Firmware Mode, it is not accessible in System Mode.

```
INVARSPEC
  FMcanAccessSegment -> !OpSMmemoryAccess
```

When a memory segment is accessible in Firmware Mode, it is not accessible in User Mode.

```
INVARSPEC
  FMcanAccessSegment ->
      !((OpProc0SegmentAccess) | (OpProc1SegmentAccess))
```

Last, we show that default values should be as restrictive as possible. For User Mode MMU tables, that means that User Mode processes should have no access rights at all after a reset.

```
INVARSPEC
  (CPU = ResetM) ->
      ((next(MMUtable[0]) = none) &
      (next(MMUtable[1]) = none))
```

Results

NuSMV [49] has been used for model checking. The model consists of 860 lines of code. 17 state variables may assume about 2^{25} distinct states. Note that only an excerpt of the variables is described here.

In the development of the formal model and preliminary runs of the model checker, nearly all proof obligations could be proven. The proof obligations that failed initially fell into two categories: Some of them identified attack paths that are available before deployment of the Target of Evaluation only. These attack paths depend on certain activities while the Target of Evaluation is in Test Mode. In practice, these attacks are not relevant, because in the deployment phase, Test Mode is no longer available. For other products, the problem of these spurious proof fails was solved by modeling the TOE life cycle explicitly. The second class of spurious proof fails was due to simultaneous access of MMU configuration parameters. Since these conditions can occur only when two processes with the same access rights interfere with each other, and since no privilege escalation may result from these racing conditions, the security policy is not violated in these cases. Therefore no redesign of the Target of Evaluation was necessary, even in the cases where proofs failed initially. In these cases additional requirements for the Target of Evaluation environment (for example, the Target of Evaluation must be completely configured before leaving the factory) suffice to satisfy all security requirements. Additional proof obligations not directly related to security functionality lead to new insights into the working of the Target of Evaluation and helped to improve the Target of Evaluation documentation.

With good documentation the modeling is quite straight forward. The scope of the model is defined by the Security Target and the level of abstraction is

mostly given by the Functional Specification. Some additional abstractions were done to simplify the model, such additional abstractions need to be analyzed to show that they do not influence the results. A discussion on the level of abstraction is given in Section 3.5.2.

To sum up, we showed that the functional specification does not allow the security IC to reach a state that is not secure, using a model checker. The model checker proved that the formal model of the IC implements the security policies. Developing a Security Policy Model requires consistent, unambiguous, and complete documentation (see Section 3.4.2). Thus, it does not only add assurance in terms of mathematical proof but also in enforcing accurate documentation.

3.4.4 Case Study - Java Card System

In this section we describe a Security Policy Model for the firewall access control policy of a Java Card OS. The Security Policy Model is compliant to the Java Card System Protection Profile [122].

A Java Card System is an operating system that runs on a security IC. It allows a developer to install and run applets on a smart card. One of the main security features of a Java Card System is the applet firewall which separates the memory of the installed applets from each other. Section 3.3.1 gives an introduction to smart cards and the Java Card OS.

In the following we first describe the formal model derived from the functional specification of the Java Card firewall [114], then we give a complete list of formal properties derived from the Security Functional Requirements of the Firewall Access Control Policy taken from [122]. Last we discuss the modeling and verification results.

Formal Model

The model of the Java Card firewall is derived from the JCRE specification [114]. All the Java Card related terms used in this section are defined in the the JCRE specification. Chapter 6 of the JCRE specification describes applet isolation and object sharing, which includes the applet firewall.

The applet firewall ensures that an applet can not access private data of another applet. To do so, every applet is assigned to an execution context, the currently active context. The JCRE is assigned to a separate context ("CurrentlyActiveContext = 0"). For every access to an object, the firewall checks if the currently active context is equal to the owner of the object. If this is not the case access is denied. Data of another applet context can only be accessed if the data has the attribute shareable. The shareable attribute makes data available to another context. It can only be set by the applet that owns the data. The JCRE can define JCRE entry point objects and global arrays which can be accessed by every applet. JCRE entry point objects are the entry points to privileged JCRE system services. They allow applets to switch to the JCRE context. Global arrays are used for data that needs to be shared across applets,

for example the communication buffer. For a more detailed description we refer to the JCRE specification [114].

Next we give a high level description of the firewall model. A schematic graph of the model is given in Figure 3.6. The model has two main states: "idle" and "locked". The "idle" state is the initial state, it represents the "normal" operating state for the JCVM. If an access is allowed by the firewall a self loop is taken and the JCVM stays in state "idle". If an access is denied by the firewall, the JCVM moves into the state "locked", which corresponds to an exception thrown by the JCRE.

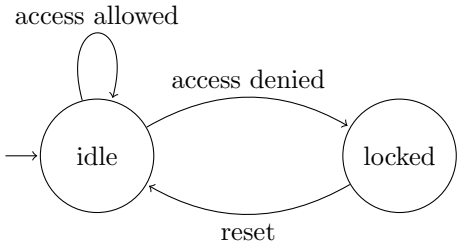


Figure 3.6: Model of the Java Card Firewall

The model captures two parts of the JCRE. First, the execution of the bytecodes with the firewall checks as described in Section 6.2.8 Class and Object Access Behavior of [114]. Second, it captures firewall related management functions of the JCRE such as context switching, object creation, and applet installation and removal.

In the following we describe the input and internal variables used in the model. The input variables are non-deterministic variables. Non-deterministic variables can take any value in the defined range at any point in time. They model the unknown behavior of the environment. For the Java Card Firewall model the input variables model the bytecodes that are executed in the virtual machine, the operations executed by the JCRE, and the object attributes that are set when creating an object. Table 3.4 describes the input variables of the model.

Variable	**Description**
bytecode[0:13]	Defines the bytecode for the firewall check, see Table 3.5.
inputJCREop	Models the management functions of the JCRE as described in Table 3.7.
inputCurrentlyActiveContext[0:2]	New current context when the current context is switched. See Table 3.8.
inputSelectedAppletContext[0:2]	New selected applet context when the selected applet context is changed. See Table 3.8.

inputIsReferenceType	If true, the field/component being stored is a reference type.
inputLCSelectionStatus	If the variable is set to "multiselectable", the object is owned by a multiselectable applet instance. If the variable is set to "nonmultiselectable", the object is owned by a non-multiselectable applet instance.
inputIsTransientObject	If true, a standard transient object is created. If false, a standard non transient object is created.
inputLifetimeTransient	If the variable is set to "clearonreset", the object is of CLEAR_ON_RESET type. If the variable is set to "clearondeselect", the object is of CLEAR_ON_DESELECT type.
inputIsArray	If true, an array is created. If false, the created object is not an array.
inputSharing	Sets the sharing parameter at creation of an object. See Table 3.6.
inputAccessShareableInterface	If true, the interface being accessed extends the Shareable interface. If false, the interface being accessed is not shareable.
inputLifetimeJCREEntryPointObject	If the variable is set to "temporary", a temporary Java Card RE Entry Point Object is created. If the variable is set to "permanent", a permanent Java Card RE Entry Point Object is created.
inputInCtxOfCurSelApplet	If false, the object is owned by an applet instance that is not in the context of the currently selected applet instance.
inputActiveOnOtherLC	If true, the object is owned by an applet instance that is active on another logical channel.
inputJCVMisRequester	If true, the JCVM is the requester for changing the currently active context.

Table 3.4: Input Variables

The bytecodes are encoded as shown in Table 3.5. Only bytecodes that require a firewall check are modeled.

Value	Description
0	no bytecode. It is used in combination with inputJCREop to perform JCRE operations like context switching.
1	getstatic
2	putstatic
3	getfield
4	putfield
5	invokevirtual
6	invokeinterface
7	athrow
8	aload
9	xastore (this stands for all variants of the bytecodes astore (bstore, iastore,...) except aastore, which is listed separately in this table)
10	arraylength
11	checkcast
12	instanceof
13	aastore

Table 3.5: Bytecodes

The "inputSharing" attribute is encoded as shown in Table 3.6.

Value	Description
sio	Sharable Interface Object
jcreentrypoint	JCRE entry point object
standard	standard object

Table 3.6: inputSharing

The "inputJCREop" is encoded as shown in Table 3.7. It models management operations of the JCRE. When no management operation is active ("inputJCREop = noop") then the model does the firewall checks for the bytecodes.

Value	Description
noop	no JCRE operation. It is used in combination with bytecode to execute bytecode.
currentlyactivecontextswitch	currently active context switch
selectedappletcontextswitch	select applet context switch
makeglobalarray	makeGlobalArray call
createobject	create object
storeobjectinval	store object in Val
installapplet1	install Applet 1
installapplet2	install Applet 2
removeapplet1	remove Applet 1

removeapplet2	remove Applet 2

Table 3.7: inputJCREop

The encoding of the context is shown in Table 3.8.

Value	Description
0	JCRE context
1	context containing Applet 1
2	context containing Applet 2

Table 3.8: inputCurrentlyActiveContext

The internal variables are set deterministically in the model. They store the security attributes of the created objects. The security attributes play a major role in the security attribute based access control policy defined in the Protection Profile. Most access control rules rely on the security attributes. Table 3.9 describes the internal variables of the model.

Object attributes	
Variable	**Description**
Owner	Owner of the object. Set at creation of the object.
isGlobal	If true, the array is designated global. Set at creation of the global array.
isJCREEntryPointObject	If true, the object is designated a Java Card RE Entry Point Object. Set at creation of the object.
LifetimeJCREEntryPointObject	If the variable is set to "temporary", the Java Card RE Entry Point Object is "temporary". If the variable is set to "permanent", the Java Card RE Entry Point Object is "permanent". Set at creation of the object.
isShareableInterfaceObject	If true, the object is a sharable interface object. Set at creation of the object.
isTransientObject	If true, the object is standard transient. If false, the object is standard non transient. Set at creation of the object.
LifetimeTransient	If the variable is set to "clearonreset", the object is of CLEAR_ON_RESET type. If the variable is set to "clearondeselect", the object is of CLEAR_ON_DESELECT type. Set at creation of the object.

isArray	If true, the object is an array. Set at creation of the object.

Table 3.9: Internal Variables

The model also includes an invariant that restricts the input variables. The invariant ensures that either a bytecode or a JCRE operation is executed in one time step.

```
INVAR
  (inputJCREop = noop | bytecode = 0)
```

The formula is part of the specification file. Note that in contrast to INVARSPEC formulas which are proof obligations, INVAR formulas are assumptions on the input variables.

Formal Properties

The Firewall Access Control Policy consists of the following Security Functional Requirements. We include all Security Functional Requirements that explicitly mention the policy and all dependencies. The dependencies are defined in the Common Criteria part 2 [37], also see Section 3.3.2 for an explanation. The list shows all the Security Functional Requirements that need to be formalized and proven on the model.

- FDP_ACC.2[FIREWALL]
  (definition of the policy, depends on FDP_ACF.1[FIREWALL])
- FDP_ACF.1[FIREWALL]
  (depends on FDP_ACC.2[FIREWALL] and FMT_MSA.3[FIREWALL])
- FDP_ROL.1[FIREWALL]
  (explicitly mentions the policy, depends on FDP_ACC.2[FIREWALL])
- FIA_UID.2[AID]
  (no dependencies)
- FMT_MSA.1[JCRE]
  (depends on FDP_ACC.2[FIREWALL], FMT_SMF.1, and FMT_SMR.1)
- FMT_MSA.1[JCVM]
  (depends on FDP_ACC.2[FIREWALL], FMT_SMF.1, and FMT_SMR.1)
- FMT_MSA.2[FIREWALL_JCVM]
  (explicitly mentions the policy, depends on FDP_ACC.2[FIREWALL], FMT_MSA.1[JCRE], FMT_MSA.1[JCVM], and FMT_SMR.1)
- FMT_MSA.3[FIREWALL]
  (depends on FMT_MSA.1[JCRE], FMT_MSA.1[JCVM], FMT_SMR.1)

- FMT_SMF.1
  (no dependencies)
- FMT_SMR.1
  (depends on FIA_UID.2[AID])

For every Security Functional Requirement of the Firewall Access Control Policy we either give a formal proof that the model satisfies the Security Functional Requirement or we give a reason why such a proof can not be given. Here we discuss the excerpt of the policy which shows the contradictions we found in the Java Card Protection Profile. The full input file to the model checker is given in Appendix A. In the following, the Security Functional Requirements are given in *italics*, the arguments or formalized properties are given in standard type setting.
FDP_ACC.2/FIREWALL
FDP_ACC.2.1[FIREWALL] The Target of Evaluation Security Functionality shall enforce the FIREWALL access control Security Function Policy on S.PACKAGE, S.JCRE, S.JCVM, O.JAVAOBJECT and all operations among subjects and objects covered by the Security Function Policy. Refinement: The operations involved in the policy are:

- *OP.CREATE,*
- *OP.INVK_INTERFACE,*
- *OP.INVK_VIRTUAL,*
- *OP.JAVA,*
- *OP.THROW,*
- *OP.TYPE_ACCESS*

FDP_ACC.2.2[FIREWALL] The Target of Evaluation Security Functionality shall ensure that all operations between any subject controlled by the Target of Evaluation Security Functionality and any object controlled by the Target of Evaluation Security Functionality are covered by an access control Security Function Policy.

The Security Functional Requirement FDP_ACC.2[FIREWALL] defines the operations of the policy. The operations are mapped to the model as follows:

- OP.CREATE is modeled by inputJCREop = createobject;
- OP.INVK_INTERFACE is modeled by bytecode = invokeinterface;
- OP.INVK_VIRTUAL is modeled by bytecode = invokevirtual;
- OP.JAVA is modeled by OP.ARRAY_ACCESS or OP.INSTANCE_FIELD or OP.INVK_VIRTUAL or OP.INVK_INTERFACE or OP.THROW or OP.TYPE_ACCESS;
- OP.ARRAY_ACCESS is modeled by bytecode = aload or bytecode = aastore or bytecode = xastore;
- OP.INSTANCE_FIELD is modeled by bytecode = getfield or bytecode = putfield;

- OP.THROW is modeled by bytecode = athrow;
- OP.TYPE_ACCESS is modeled by bytecode = checkcast or bytecode = instanceof.

FDP_ACF.1/FIREWALL
FDP_ACF.1.1[FIREWALL] The Target of Evaluation Security Functionality shall enforce the FIREWALL access control Security Function Policy to objects based on the following:

Subject/Object	*Security attributes*
S.PACKAGE	*LC Selection Status*
S.JCVM	*Active Applets, Currently Active Context*
S.JCRE	*Selected Applet Context*
O.JAVAOBJECT	*Sharing, Context, LifeTime*

The Security Functional Requirement FDP_ACF.1.1[FIREWALL] defines the subjects, the objects and the security attributes of the policy. They are mapped to the model as follows:

- S.PACKAGE is modeled by CurrentlyActiveContext != 0;
- S.JCVM is modeled by inputJCVMisRequester;
- S.JCRE is modeled by CurrentlyActiveContext = 0;
- S.PACKAGE.LC Selection Status is modeled by inputLCSelectionStatus;
- S.JCVM.Active Applets is modeled by inputActiveOnOtherLC;
- S.JCVM.Currently Active Context is modeled by CurrentlyActiveContext;
- S.JCRE.Selected Applet Context is modeled by SelectedAppletContext;
- O.JAVAOBJECT.Sharing is modeled by isGlobal or isJCREEntryPointObject or isShareableInterfaceObject, set by inputSharing at creation of an object;
- O.JAVAOBJECT.Context is modeled by Owner;
- O.JAVAOBJECT.LifeTime is modeled by isTransientObject and LifetimeTransient.

FDP_ACF.1.2/FIREWALL The Target of Evaluation Security Functionality shall enforce the following rules to determine if an operation among controlled subjects and controlled objects is allowed:

- *R.JAVA.1: S.PACKAGE may freely perform OP.ARRAY_ACCESS, OP.INSTANCE_FIELD, OP.INVK_VIRTUAL, OP.INVK_INTERFACE, OP.THROW or OP.TYPE_ACCESS upon any O.JAVAOBJECT whose Sharing attribute has value "JCRE entry point" or "global array".*

First we show the property for the four operations OP.INVK_VIRTUAL, OP.INVK_INTERFACE, OP.THROW, and OP.TYPE_ACCESS. We show that S.PACKAGE (CurrentlyActiveContext != 0) can invoke the given bytecodes on a JCRE entry point object or a global array without moving to the "locked" state. Thus the operations are allowed.

```
LTLSPEC
  G(CurrentlyActiveContext != 0 & (isJCREEntryPointObject
      | isGlobal) & ((bytecode >= 5) & (bytecode <= 7) |
    (bytecode >= 11) & (bytecode <= 12)) & (state = idle
    ) -> X(state = idle))
```

For OP.INSTANCE_FIELD we can not prove the requirement. The specification for "getfield" and "putfield" (Section 6.2.8.3 Accessing Class Instance Object Fields of [114]) does not allow access to JCRE entry point objects or global arrays. We can prove the following property on our model. The model moves into state "locked", meaning that the access is not allowed.

```
LTLSPEC
  G(CurrentlyActiveContext != 0 & (isJCREEntryPointObject
      | isGlobal) & (bytecode = 3 | bytecode = 4) & (
    state = idle) -> X(state = locked))
```

For OP.ARRAY_ACCESS the requirement can only be partly shown. We can show the property for global arrays for the bytecodes "aload" and "xastore".

```
LTLSPEC
  G(CurrentlyActiveContext != 0 & isGlobal & (bytecode =
    8 | bytecode = 9) & (state = idle) -> X(state = idle
    ))
```

The specification for "aastore" (Section 6.2.8.2 Accessing Array Objects of [114]) does not allow access to global arrays. The following property can be proven on our model.

```
LTLSPEC
  G(CurrentlyActiveContext != 0 & isGlobal & bytecode =
    13 & (state = idle) -> X(state = locked))
```

The JCRE specification requires for "astore" and "aload" that the accessing context is either the JCRE or the owner of the object if the object is not a global array. Only the JCRE can designate JCRE entry point objects. Thus there are no JCRE entry point objects that do not belong to the JCRE. The following property can be proven on our model.

```
LTLSPEC
  G(CurrentlyActiveContext != 0 & isJCREEntryPointObject
    & !isGlobal & (bytecode = 8 | bytecode = 9 |
    bytecode = 13) & state = idle -> X(state = locked))
```

This concludes our list of properties for R.JAVA.1. We show that the property is not true for OP.INSTANCE_FIELD and that the property is only partly true for OP.ARRAY_ACCESS. We propose to remove OP.INSTANCE_FIELD and OP.ARRAY_ACCESS from rule R.JAVA.1 and add separate rules for these operations.

- *R.JAVA.5: S.PACKAGE may perform OP.CREATE only if the value of the Sharing parameter is "Standard".*

First we show that S.PACKAGE may perform OP.CREATE if the value of the Sharing parameter is "Standard". We use CTL to show that there is a path which allows access, thus S.PACKAGE may perform OP.CREATE.

```
CTLSPEC
  EF(CurrentlyActiveContext != 0 & inputJCREop =
      createobject & inputSharing = standard & state =
      idle & EX(state = idle))
```

Next we try to prove the only if direction of the requirement with the formula given below. But the proof fails because S.PACKAGE is allowed to e.g. create a shareable interface object (Section 6.2.4 Shareable Interfaces of [114]).

```
LTLSPEC
  G(CurrentlyActiveContext != 0 & inputJCREop =
      createobject & state = idle & X(state = idle) -> (
      inputSharing = standard))
```

The Protection Profile provides a footnote for the operation OP.CREATE stating that shareable transient objects are not allowed. We prove this on our model with the following property.

```
LTLSPEC
  G(isShareableInterfaceObject -> !isTransientObject)
```

This concludes our proof of R.JAVA.5. We show that the 'only if' direction can not be proven. We assume that the rule was supposed to state that shareable transient objects are not allowed, which we prove with the last formula given above. We propose to change R.JAVA.5 in the Protection Profile.

FMT_MSA.3[FIREWALL]

FMT_MSA.3.1[FIREWALL] The Target of Evaluation Security Functionality shall enforce the FIREWALL access control Security Function Policy to provide restrictive default values for security attributes that are used to enforce the Security Function Policy.

At creation of an object the context attribute is set to the creators context.

```
LTLSPEC
  G(inputJCREop = createobject & state = idle & X(state =
      idle) -> X(Owner = CurrentlyActiveContext))
```

FMT_MSA.3.2[FIREWALLEditoriallyRefined] The Target of Evaluation Security Functionality shall not allow any role to specify alternative initial values to override the default values when an object or information is created.

We show this by claiming that there exists a path where an object is created with a context attribute different to the creators context, see property below. The property is proven false, thus we show that all objects are created with the creators context as the objects context.

```
CTLSPEC
  EF(inputJCREop = createobject & state = idle & EX(state
      = idle) & EX(Owner != CurrentlyActiveContext))
```

Note that the last two formal properties are semantically equivalent. They look different because they are translated from two different properties. With the abstraction level of our model we can not distinguish between the two Security Functional Requirements. In essence we can only show that the context attribute is always set to the creators context at creation of an object.

Results

The model is generated with COSIDE, which translates an UML statechart diagram into NuSMV model. The diameter of the NuSMV model is 17. The model has roughly 2^{35} reachable states out of roughly 2^{53} states. The NuSMV model checker is used for verification.

The verification only takes a few seconds. It shows that all properties except for two are true. Both failing properties are discussed above. Not all false properties must necessarily point to a contradiction between requirements and implementation. Failing properties can be used to show that a certain path does not exist. Still, there are some Security Functional Requirements that we could not prove, in fact we proved that parts of the given Security Functional Requirements are not true for our model. We claim that two rules of FDP_ACF.1.2/FIREWALL contradict the Java Card specification [114]:

- R.JAVA.1 for OP.INSTANCE_FIELD and OP.ARRAY_ACCESS, and
- R.JAVA.5.

We propose to refine the contradicting rules in the Protection Profile. A first draft of the new Protection Profile already contains refined and new rules to remove the contradictions.

Again we see that informal specifications and semi formal requirements can be ambiguous and misleading. The formal model uncovers imprecise requirements that contradict the specification.

3.5 Modeling using UML Statecharts

The above described method is an efficient method for security policy modeling that satisfies the Common Criteria requirements. But to leverage all the advantages of applying formal methods in an early design phase, for example the specification phase, the method needs to be integrated into the design process of

a product. In order to do so the benefits must be clear to all parties in a project. To apply formal methods in an early design phase has numerous advantages apart from satisfying Common Criteria requirements:

1. Generation of a *precise understanding of the specification*, since there is no room for interpretation in a formal model.

2. Improvement of the *documentation*. The model and properties formally link the functional specification to the requirements and offer a consistent and unambiguous documentation.

3. *Mathematical proof* that the functional specification satisfies the requirements.

Avoiding errors early and a clear communication of the specification to the engineer are key features for a short time to market and high quality products.

The next section gives a short description of how to integrate formal modeling of specifications into the design process. Different levels of integration are described. Section 3.5.2 details the possibilities of formally modeling specifications. Not every part of a specification is suitable for being modeled and for efficiency it is necessary to find the right level of abstraction. The last section shows an example model using UML statecharts.

3.5.1 Integration into the Design Process

The least impact on the design process is achieved if the formal verification is done separately. After the requirements have been written down and the specification has been finished, the formal verification engineer gets the documents as input to carry out the formal verification. If the model is written in an uncommon language like NuSMV [48], the model can not even be reviewed by the architect who is responsible for the specification. Apart form the work presented here, all the published formal verification done for Common Criteria in the smart card industry is separated form the design process of the product (see Section 3.6.1).

A first step towards integrating the formal verification into the design process is to use UML statechart diagrams as modeling language. This enables the architect to review the model and it eases the communication between the architect and the verification engineer. A second step is to use UML statecharts as specification language. This requires training of the architect who is responsible for writing the specification to generate awareness of the restrictions of a model checker.

The first two steps allow us to integrate the modeling part into the design process. The second input to the model checker, the properties, are derived by formalizing functional and security requirements. Currently, they have to be translated manually to temporal logic formulas. For future work, we plan to offer templates to allow non-experts in formal methods to formalize requirements.

The formal modeling and formal verification of abstract specifications is a team effort. If the model is required for Common Criteria certification, the

Common Criteria evaluation engineer defines the scope of the model and the requirements that need to be proven, the (security) architect defines the model and the formal methods expert supports in formalization of the requirements and model.

3.5.2 Level of Abstraction

The main limitation of model checking is the so called state space explosion problem. The size of the state space is exponential in the number of variables defining the states. Thus, large numbers and big data structures should be avoided. Only being able to deal with a finite state space and considering the state space explosion problem entails the following restrictions:

- It is not feasible to model all physical properties of a hardware. Consequently, it is not state of the art to formally verify at the physical level if a system is resistant to physical attacks such as inherent information leakage (for example power analysis attacks), physical probing, malfunction due to environmental stress, or physical manipulation.
- It is not feasible to model complex algebraic functions. Thus, it is not state of the art to formally verify if a cryptographic function like a random number generator, encryption, or decryption works correctly.

The level of abstraction of a formal model is not only restricted by the state space explosion problem. The following considerations should also be taken into account. First, it is only necessary to model details that are relevant to prove interesting requirements. We do not want to crowd the model with unrelated details. Second, if the model has too much detail, it may become too complex. The complexity of the model may make the results of the proofs hard to understand and hence make the model less useful.

The guideline for formal security policy modeling for Common Criteria [3] (Section 5.2 *Determination of the Degree of Abstraction*) states "Generally finding the right abstraction degree can not be done schematically. A creative process is needed during which such a construction is usually adapted several times". The following examples show some possible abstractions:

- Often an induction argument can be used to only model a small number of a possible large number of items. For example, if a smart card can have a large number of similar applications installed, and the security requirement is not influenced by the interaction of the applications, then it is sufficient to only model one application, see Section 3.5.3 for an example. In case the security requirement to be proven concerns the interaction of the application, it may be sufficient to model two applications and argue that the requirement holds for all applications using induction.
- Usually data structures are not modeled because they do not contribute to the logical behavior of a system. Modeling data structures leads to an enormous increase in the state space but since they do not contribute to

the logical behavior of a system, they do not change the verification result, and thus add no value to the model. In our example (Section 3.5.3) we do not model the data structure of the keys, the commands, the application, the memory, etc.

- In most models it is helpful to carry out some additional abstractions on the functionality. In our example (Section 3.5.3), we only modeled the increment and decrement of the value by 1. Extending the model to allow an increment or decrement up to the maximal respectively minimal value is trivial, but it would only make the model more complex without adding any real new value with respect to the requirements we want to prove.

Finding the right level of abstraction is one of the most challenging tasks of applying formal verification. If the model is too abstract, it will not help to improve the quality of the product. Having a too detailed model is not efficient. The optimum between vacuity and effectiveness has to be assessed for every model individually. Naturally for security relevant parts or complex functional parts of a product, having a more detailed model pays off. Modeling straight forward functionality may make sense on a higher level of abstraction in order to have an unambiguous specification.

Having explained the limitations of model checking and the challenge of finding the right level of abstraction for a model, we conclude this section by describing what can be modeled. Our method is tailored to modeling complex behavior and security relevant functionality. We use it to model security policies, especially access control policies, see Section 3.5.3.

3.5.3 Example

In this section we present an illustrative example modeling a simplified access control policy specification as it could be implemented for a smart card. For smart cards deployed in security critical applications, such as banking (e.g. ATM card, credit card), e-government (e.g. health card, passport) and access control (e.g. access to buildings, transportation) it is important to ensure that only authorized users have access to certain functionality. For example, only users that know the PIN are allowed to withdraw money with an ATM card; only an authorized administrator can write data onto a passport (generate a passport); only an authorized person is allowed to read data from a passport.

The model we present here is a simplified version of the DESFire Access Control Policy as defined in [111]. The original model has a system diameter of 14, it has roughly 2^{74} reachable states out of roughly 2^{96} states. We defined 146 properties which take about 30 minutes to prove. As a comparison, the model we present here has a system diameter of 16, it has roughly 2^{11} reachable states out of roughly 2^{18} states and the verification of the properties takes about 2 seconds.

The simplified model we present here has a card manager that is able to create and delete application(s). For example, a public transport company can create an application on the card to allow a customer to load the card and then use the card as a ticket. For this example we only model one application for

simplification without loss of generality since the properties we want to prove do not depend on the interaction between applications. The application we model has a value that can be incremented and decremented. Considering a public transport card again, the increment and decrement functions of the application can be seen as an abstraction of the loading functionality and the ticket use.

We use the following notation: Capital letters are used for input variables. Variables with lower case letters are internal or output variables.

Figure 3.7 shows the overview state diagram of our formal system model. The initial state is the "Idle" state. From this state it is possible to either select the card manager or the application, if it is active. The card manager is chosen to manage the card, to delete or create applications. The application is chosen to use the application, for example load or use the card.

The card manager and the application are each modeled in a substatechart. Figure 3.8 and Figure 3.9 show the statecharts of the card manager and the application, respectively. If the card manager is selected (the state "CardManager" is active) then the statechart moves into the "CM_NotAuthenticated" state of the substatechart. By issuing the "AUTHENTICATE" command with "KEY== 0", it is possible to authenticate the card manager. The card manager is then allowed to create or delete an application. To create an application, the "CREATE" command and the number of the application to be created ("APP==1") has to be issued to the card. Then the application is set to active and the value is set to 0. Deleting an application works similarly. If neither the delete nor the create transition are triggered, then there is an error and the authentication state is lost.

The order in which the transitions are tried is given by the priorities of the transitions. In this case the create transition has priority 1 and the delete transition has priority 2. Since their guards are mutually exclusive the priorities do not have an impact on the taken transition. But the transition with priority 3 has no guard and the least priority, thus it is taken in all cases not covered by the other two transitions.

The application statechart is similar to the card manager statechart, except that the application user can increment or decrement the internal variable "value" between 0 and 5.

Beside the state diagram, a specification file is needed as input to COSIDE. In this file, the CTL and LTL properties, and the restrictions on the input variables are given. Every property has to be preceded by the formula type "CTLSPEC" or "LTLSPEC". These formulas are a translation of the requirements of the security policy we want to prove. CTL and LTL are temporal logic languages that are used to formalize the temporal properties of systems (see Section 3.3.3).

Assumptions on the input variables are preceded by the keyword "INVAR". We use these invariants to restrict the input variables of the model. For example, an assumption has been added to model that it is only possible to send one command to the card in each time step. The "RESET" signal is an exception because a reset can occur anytime, when the card is removed from the reader. The following invariant describes this behavior and is copied from our specification

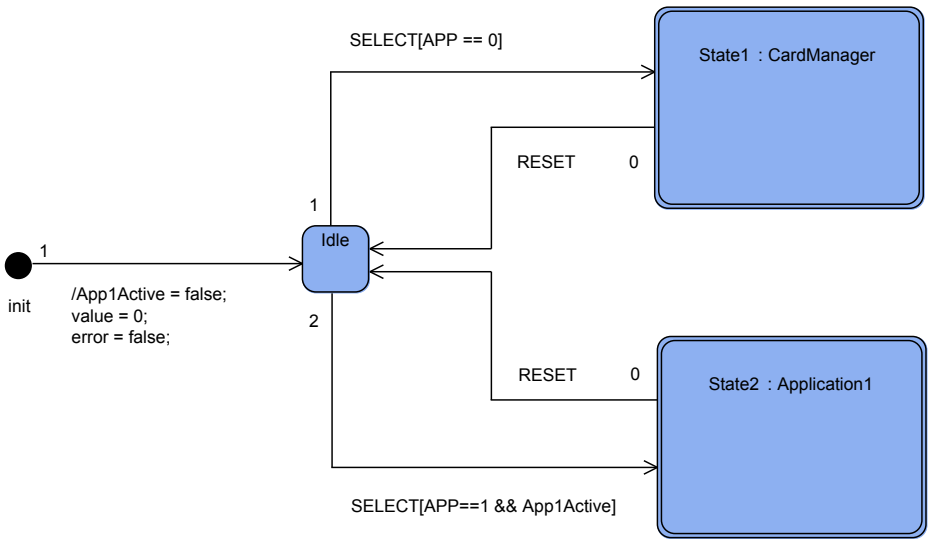


Figure 3.7: Access Control Policy Overview state diagram

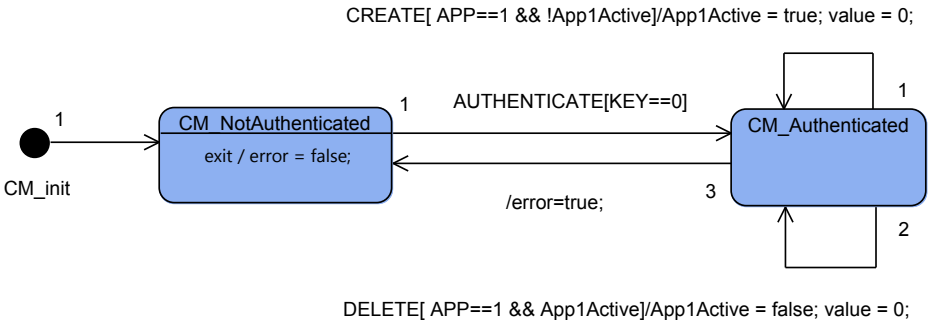


Figure 3.8: Access Control Policy Card Manager state diagram

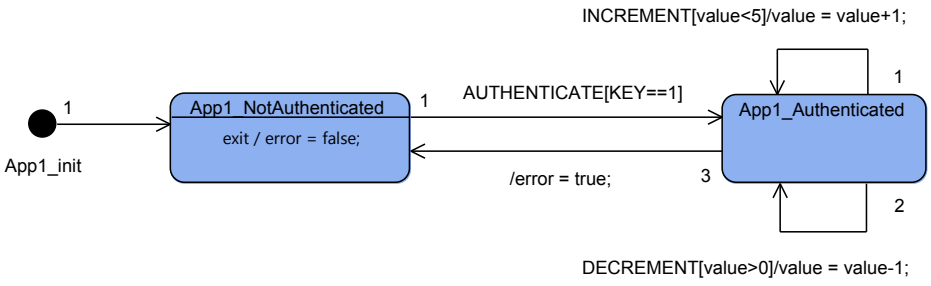


Figure 3.9: Access Control Policy Application1 state diagram

input file. The function "count(var1, var2, ...) = 1" is true if and only if exactly one of the variables "var1, var2, ..." is true.

```
INVAR
count(SELECT, AUTHENTICATE, CREATE, DELETE, INCREMENT,
    DECREMENT) = 1
```

First we prove that it is only possible to create or delete an application if authenticated with the card manager key ("KEY==0"). We only show the formulas for creating, the formulas for deleting are similar. The property is build up in three steps with the three SPEC formulas given below. The first formula shows that the right hand side of the implication in the third formula is not false. If the right hand side of an implication is false then the implication is vacuously true which could lead to wrong conclusions. The second and the third formula together give the property we want to show.

```
// A path exists (E) such that eventually (F),
// App1Active is false and
// in the next state (X) App1Active is true
// - thus it is possible to create an application
CTLSPEC
EF(!App1Active & EX App1Active)

// Always (G),
// when the state changes to CM_Authenticated
// then the authentication key is the card manager key (
    KEY=0)
// - thus state=CM_Authenticated -> authenticated with
    the card manager key
LTLSPEC
G(state!=CM_Authenticated & X(state=CM_Authenticated) ->
    KEY=0)

// Always (G),
// when an application is created
// then the state is CM_Authenticated
// - thus it is only possible to create an application
    when authenticated with the card manager key
LTLSPEC
G(!App1Active & X(App1Active) ->
  state=CM_Authenticated)
```

The next two formulas check that the authentication state is lost when an error occurs.

```
// Always (G),
// when error is true
// then the state is not CM_Authenticated
```

```
LTLSPEC
G(error -> state!=CM_Authenticated)

// Always (G),
// when error is true
// then the state is not App1_Authenticated
LTLSPEC
G(error -> state!=App1_Authenticated)
```

Last we want to show that it is only possible to increment or decrement the value if authenticated with the application key. Again we try to show this in three steps. First we prove that the value can be changed. Again this property is necessary to ensure that the right hand side of our implication in the third formula is not false.

```
// A path exists such that eventually,
// the value is 0 and in the next step the value is
// not 0 or ... (the same for all values up to 5)
// - thus it is possible to change the value
    (increment or decrement the value)
CTLSPEC
EF(value=0 & EX(value!=0) | value=1 & EX(value!=1) |
   value=2 & EX(value!=2) | ... )
```

Next we show that state "App1_Authenticated" can only be reached when authenticated with the application key.

```
// Always (G),
// when the state changes to CM_Authenticated
// then the authentication key is the application key (
   KEY=1)
// - thus state = CM_Authenticated -> authenticated with
   the application key
LTLSPEC
G(state!=App1_Authenticated & X(state=App1_Authenticated)
    -> KEY=1)
```

The next formula does not hold. It cannot be shown that the value can only change when authenticated with the application key. The model checker returns false and a counter example.

```
// Always (G),
// when the value changes
// then the state is App1_Authenticated
// - thus the value can only change when authenticated
   with the application key
LTLSPEC
G( (value=0 & X(value!=0) | value=1 & X(value!=1) |
    value=2 & X(value!=2) | ... ) ->
```

state=App1_Authenticated)

COSIDE offers the possibility to visualize counter examples on the model. See Figure 3.10 for a screen shot of the visualization of the counter example. The visualization shows the currently active state as well as the transitions that were taken before. Moreover, transitions can be step-wise executed forward and backward to ease debugging.

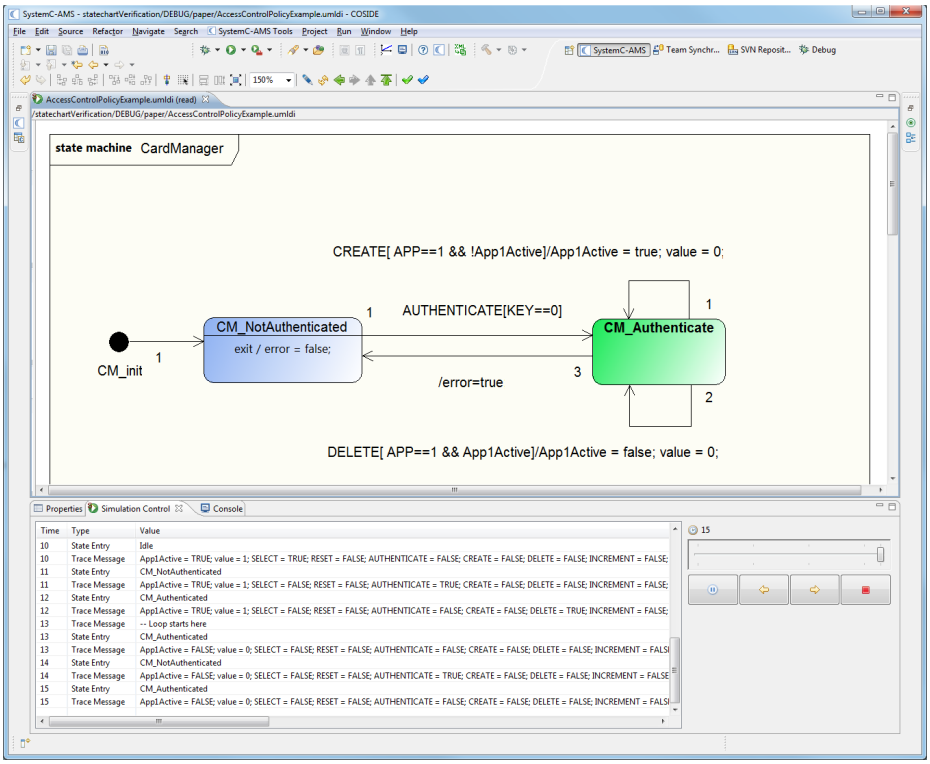


Figure 3.10: Visualization of counter example

The visualization shows that deleting the application can also change the value. Thus the value can also change when not authenticated with the application key. Since the deletion requires authentication with the card manager key, this behavior is not a security risk. With this information the specification can be improved and the following formula can be derived, which holds on the model.

```
// Always (G)
// when the value changes
// then the state is App1_Authenticated or
// CM_Authenticated
// - thus the value can only change when authenticated
    with the application key or the card manager key
```

```
LTLSPEC
G( (value=0 & X(value!=0) | value=1 & X(value!=1) |
    value=2 & X(value!=2) | ... ) ->
    (state=App1_Authenticated |
     state=CM_Authenticated))
```

The above example shows how the specification can be debugged using the counter examples of the model checker. Next we give an example showing how the model checking fails if the model is not correct. Assume the guard on the authentication transition of the card manager is missing, see Figure 3.11.

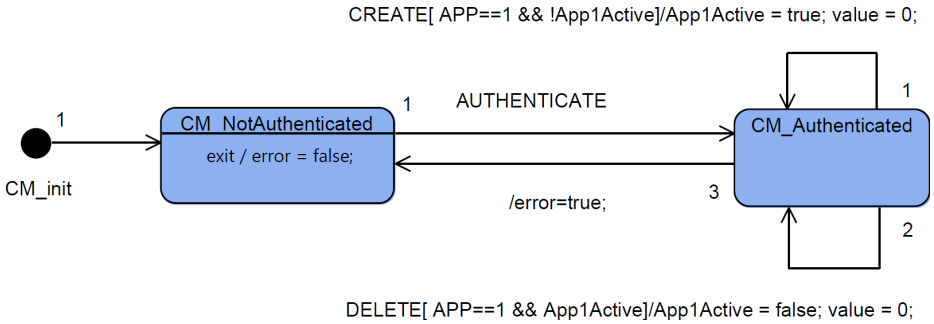


Figure 3.11: Access Control Policy Card Manager state diagram without key check

This is a big security issue since the authentication state can be reached without the correct key. When we run the model checker on the modified model the following property fails.

```
// Always (G),
// when the state changes to CM_Authenticated
// then the authentication key is the card manager key (
    KEY=0)
// - thus state=CM_Authenticated -> authenticated with
    the card manager key
LTLSPEC
G(state!=CM_Authenticated & X(state=CM_Authenticated) ->
    KEY = 0)
```

The counter example shows that a switch to state "CM_Authenticated" is possible with "KEY = 1". With this knowledge it is easy to find the bug in the model and correct it.

The generation and visualization of counter examples can help in two ways. For formulas that we expect to be true, but which are false, the counter example can help to understand the problem and find a solution. Additionally the visualization can also be used to check if the model works as expected. For example, if we want to see how the state "App1_Authenticated" can be reached we add the formula given below. The visualization of the counter example will show how the state can be reached.

```
// Always (G) state is not App1_Authenticated
// - thus the state App1_Authenticated can never be
    reached
LTLSPEC
G(state!=App1_Authenticated)
```

The example given in this section is only a very small model but it demonstrates the current possibilities of model checking UML statechart diagrams with COSIDE. Being able to model with a widely known and understood language is a first step towards integrating formal verification of specifications into an industrial design process.

3.6 Related Work

We split the related work into four categories. First we describe work related to Security Policy Models according to Common Criteria. Then we describe work related to model checking and work on integrating formal methods into the development process. The last section describes related work on formal verification of UML statecharts.

3.6.1 Common Criteria

The first Evaluation Assurance Level 6 certificate was received by STMicroelectronics for the ST23YR80. The Dynamic Memory Access Control Policy of the security IC was modeled. It was certified by the French certification body [1]. Unfortunately no publications can be found describing the used method.

Most of the publications describing a Security Policy Model for Common Criteria use theorem provers. Below we give examples for four different theorem provers: ACL2, Isabelle, Coq, and PVS.

ACL2 The ACL2 theorem prover was used to develop and prove a formal security policy model for the INTEGRITY-178B real-time operating system [107]. The main policy proven for INTEGRITY-178B was a policy about separating operating system kernels called "GWV policy" [85], which is very similar to the Access Control Policy described in Section 3.4.3.

The ACL2 theorem prover was also used in the evaluation of the Rockwell Collins' AAMP7G microcontroller to show that the microcontroller satisfies the GWV kernel separation criteria [124]. In the AAMP7G project two different abstraction layers were modeled: the formal abstract model and the formal detailed model. First a proof is given showing that the abstract model satisfies the security policy, then a proof is given showing that the detailed model is a refinement of the abstract model. Eventually a detailed code to model review is conducted to guarantee that the implementation corresponds to the detailed model. ALC2 is well fitted for such a layered approach, but the additional layer requires additional modeling and proving effort. This additional effort is not necessary for certification, see discussion in [123].

With our approach we only model one abstraction layer and prove that the model satisfies the security policy, which is less effort and satisfies the Common Criteria requirements for high evaluation assurance levels. Note that the work in [123] refers to Common Criteria version 2.1 whereas our work refers to Common Criteria version 3.1. Version 2.1 requires a high level and a low level design which are merged into one required design in version 3.1. In [123] the authors give a high level description how Common Criteria high evaluation assurance levels can be reached with ALC2. We show how Common Criteria high evaluation assurance levels can be reached with model checking, giving very detailed rationales in Section 3.4.2.

Our approach is tailored to efficiently develop an abstract Security Policy Model for Common Criteria. Compared to our approach, the work using ALC2 is much more elaborate, requiring a big effort to manually guide the proofs. The paper [124] states that "Nearly all the time on the project was spent constructing the proofs ...". This is not required when a model checker is used.

Isabelle The Isabelle theorem prover was used to verify the security model of Infineons SLE 88 Smart Card Memory Management [136]. The security model uses Interactive State Machine (ISM)s to model the memory management of the SLE 88. From a conceptual point of view the security model of the SLE 88 is quite similar to our model of the access control policy of the NXP Secure Smart Card Controller P60x144/080PVA given in Section 3.4.3. Also, ISMs can be defined graphically using the Computer Aided Software Engineering (CASE) tool AutoFocus. Using a converter, the graphical representation can be translated to the Isabelle/HOL representation. The main difference to our approach is that the proofs conducted by Isabelle need manual guidance, for example invariants need to be defined and proven and assumptions need to be defined. The article [135] states "Much more involved is the proof of our final theorem ...: we need an invariant .... This in turn requires two assumptions ... such that the invariant is maintained". We argue that our model checking push-button approach is much more user friendly and the additional expressive power provided by Isabelle is not necessary for Security Policy Modeling. The conclusions of the Security Policy Model work in the article [135] are quite similar to our results provided in Section 3.4.3, the model provides new insights in the working of the Target of Evaluation and helps identify the relevant assumptions on the system environment.

Coq The theorem prover Coq was used for security policy modeling of the Java Card firewall [46]. Chetali and Nguyen propose a method that satisfies the requirements of the formal components for Evaluation Assurance Level 7 of Common Criteria version 2.2 (see Security Target [78]). For the Security Policy Model the security objectives stated in the Security Target are formalized into properties represented as Firewall Virtual Machine state machine, and the Security Functional Requirements are formalized into the Security Policy Model represented as JCVM state machine. The Firewall Virtual Machine state machine is based on Chapter 6 of [114] like our formal model given in Section 3.4.4. The JCVM state machine models the card state which is composed of the installed applets, the runtime data (frame, stack, heap), the static data, and others. A

refinement proof between the Firewall Virtual Machine state machine and the JCVM state machine is given to show that the JCVM satisfies the firewall rules. This shows that there can be quite different interpretations of the Common Criteria assurance requirements for the Security Policy Model.

The link to the Functional Specification is formal since the work targets Evaluation Assurance Level 7. The Functional Specification and High Level Design are represented as Formal Internal Virtual Machine state machine and another refinement proof is given to show that it satisfies the firewall rules. Like in the work using ALC described in [123] refinement proofs are given down to the Low Level Design. The last step from the Concrete Virtual Machine state machine which is the most detailed formal representation to the implementation is done by manual review as in [123].

Compared to our work the work described in [46] was a much bigger effort but the target was also a different one, aiming for Evaluation Assurance Level 7 in Common Criteria version 2.2 compared to Evaluation Assurance Level 6 in Common Criteria version 3.1. Independent of the different target our approach has the advantage of automated model checking. The authors of [46] mention that about 10% of the proofs needed user-defined tactics and that maintenance of these proofs has high costs. Both user defined proofs and the related maintenance can be avoided with model checking. Additionally we think that it is not efficiently feasible for a non expert to review/understand the model used in [46] which is certainly possible with our UML modeling approach described in Section 3.5.

PVS The theorem prover PVS was used in [88] for the Common Criteria certification of an Embedded System. The front end TAME [7] was used to formalize the top level specification of the separation kernel of the Embedded System. TAME is a tool that translates state machine models into PVS and offers proof support. A data separation policy is proven for the model of the separation kernel. The main effort on this part of the project was spent on conducting the proofs which can be avoided with our model checking approach.

The authors of [88] emphasize that for communication between the formal methods team and the development team it is necessary to have a representation of the top level specification that can be understood by all parties. They use a natural language representation. In Section 3.5 we propose to use the UML statecharts as common representation. This avoids maintenance effort to keep the formal and the natural language representation synchronized.

Apart from the theorem provers different specification languages were proposed. The two most popular specification languages used for security policy modeling are B and Z. We give some examples below.

B The B Method is based on refinement, an abstract model is refined into a more detailed model, possibly down to the implementation. The correctness of the refinement step is usually proven with a theorem prover. Motre and Teri [109] propose to use UML for an informal model and the B Method for the formal model of a high assurance level Common Criteria evaluation. They evaluate their method on a model of the JCRE. One big advantage of our approach is that we do not have an informal model but use the UML model directly for the Security

Policy Model.

Z Z is a similar specification language to B. Morimoto et al. [108] propose to use Z for formalization and verification for security specifications based on common criteria. Z is also used in the work of Hall and Chapman [87], where they propose a development process based on formal methods. Requirements from certification schemes are one motivation to use formal methods but the main goal of the work described in [87] is to reduce development costs especially in costly bug fixing.

Generally, Common Criteria certifications are not the main motivation to formalize a specification. Below we give some examples of publications describing formalizations of parts of the Java Card Specification. Although their main goals are different they argue that their work is suitable for Common Criteria evaluations. Examples are

- The formalization of the Virtual Machine and the firewall with Coq presented in [11]. The main goal of this work is to formally verify security properties of applets.
- The formalization of the bytecode verifier with Isabelle/HOL presented in [99]. They prove the correctness of the bytecode verifier, which again guarantees security properties of the applets.
- The formalization and verification of the GlobalPlatform Card Specification using the B Method [13].

All the above referenced methods for high assurance level Common Criteria certifications require an interactive theorem prover to prove the security policies. In contrast we propose to use model checking. We argue that model checking is more efficient because the input language is easier to understand for non experts and the mathematical proofs are done fully automatic.

3.6.2 Model Checking

We use the model checker NuSMV. NuSMV has been used in a number of projects to proof security and safety properties. For example in formal modeling the FCS 5000 flight control system and the ADGS-2100 Adaptive Display and Guidance System, Operational Flight Program (OFP) [107]. Another example is the work of Chan et al. [40] on model checking large software specifications, where they verify a state-based system requirements specification of a Traffic Alert and Collision Avoidance System II (TCAS II). Chan et al. describe a translation form statecharts to SMV based on the TCAS II example and show that is is possible to prove different properties for that specification. Conceptually their work is quite similar to ours but since their publication in 1998 the tooling improved considerably in power and automation. COSIDE translates statecharts directly to NuSMV and specifications of considerable size can be verified in a few minutes. In contrast to their work, our main contribution is to show how model checking can be used for Common Criteria Evaluation Assurance Level 6 certifications and how model checking specifications can be integrated into the design process.

NuSMV is also used to model check Formal Tropos Specifications in the T-Tool. The tool is build on a software methodology for model checking early requirements specifications [77]. The proposed methodology seems close to the method we propose, but it is tailored to agent-oriented software engineering, concentrating on understanding the environment of the software.

In [104], the usability of NuSMV is analyzed and a tool is proposed to improve the usability of the model checker with the goal to integrate it effectively into the development process. Unfortunately the tool never seemed to reach industrial level, but we follow their advice and use a graph based input language for modeling.

3.6.3 Model Driven Engineering

In Section 3.5 we make a proposal on how to integrate model checking into the design process of a smart card. Also the work in [137] describes how to integrate formal analysis into a model-based software development process. They argue that formal verification of the model should be done at the beginning of the development process to find bugs early in the project and save costs in bug fixing later. They also give guidelines on where to use formal and how. The paper states that "To use formal methods most effectively, some changes must be made to the traditional development cycle, and formal analysis should be considered when creating requirements and designing models". We agree with these statements; to get the most benefits out of formally analyzing requirements and specifications, the formal modeling and analysis should be considered as part of the development process from the start of the project. In contrast to our work, the proposal of [137] is based on Simulink/StateFlow models used in the avionics industry. They verify functional and safety requirements, where we verify security functional requirements for Common Criteria certifications of smart cards.

The SPES modeling framework [30] is a more recent model-based development framework that supports the use of models as main development artifacts in all phases of the development process. It deals with a broad range of application fields (automation, automotive, avionics, energy, and healthcare) and a broad range of viewpoints (requirements, functional, logical, technical) for system modeling. Campetelli et al. [34] apply the model based SPES development method to an industrial case study from the automation domain. Similar to our work they formally verify a specification against requirements in an early development phase. They also aim for integration of modeling and automatic verification with the development process.

Modeling and verification of the specification is only a first step towards model driven engineering. One of the main drawbacks of our method is the missing formal link to the actual implementation. Such a formal link is not required by Common Criteria [12]. We assume that formal verification of the implementation is not required by Common Criteria because for many products it might not be feasible. Still such a link would be very useful to increase the confidence that the implementation satisfies the requirements.

Several works propose to establish a link between the formal model and the implementation by generating test cases from the formal model and to run them on the implementation e.g. [45], [31], [28], and [29]. In [28] we generate test cases for the Java Card firewall model described in Section 3.4.4. The generated tests increased the test coverage of the standard Java Card test suite (JCTCK) and uncovered an inconsistency between the tested implementation and the modeled specification. An interesting discussion on model-based test case generation for smart cards can be found in [117].

For some products not only test case generation is feasible but also code generation. For example for logic controllers. The work presented in [86] proposes to synthesize logic from a formal model of a controller based on UML activity diagrams. For smart cards there are two main obstacles for automatic code generation. First, code has to be highly optimized in size, speed, and power consumption. Second, for security products countermeasures against hardware attacks have to be implemented. Both are interesting fields for further research.

Some projects even establish a formal link between the formally verified specification and the implementation. For example the seL4 OS Kernel project [98]. First, they formally verify that the abstract model of the specification satisfies the functional properties. Then, they give a formal refinement proof from the abstract model of the specification down to the C implementation of the microkernel. The theorem prover Isabelle/HOL is used for the formal proofs. This is the most rigorous approach with respect to formal verification in model driven engineering that has been applied in an industrial sized project. It is also the approach which requires the greatest changes in the design process and thus is the most risky. Our approach described in Section 3.5 only requires small changes in the design process and is thus more likely to be accepted as a first step towards the integration of formal verification in an industrial project.

3.6.4 Formal Verification of UML Statecharts

The idea to use a graphical representation of the specification to analyze and formally verify the specification is already more than 20 years old [89]. Helbig et al. [89] provide a development environment for VHDL designs based on timing diagrams, statecharts, and temporal logic. Both VHDL and statecharts are translated into transition systems and verified against timing diagrams and temporal logic formulas by a model checker. Our approach operates at a higher level of abstraction using UML statecharts.

In this section we give a few examples of different tools that use model checkers to verify UML diagrams.

HUGO [128] is a prototype tool that uses the model checker SPIN [93]. HUGO can check whether the interactions expressed by a UML collaboration diagram are realized by a set of state machines. Unfortunately, the tool is not mature enough to be used in an industrial project.

vUML [103] is another tool using the model checker SPIN. vUML detects common design errors in UML state diagrams such as deadlocks, reaching an invalid state, constraint violations, and more. Apart from the fact that the set of

properties that can be verified is limited, the major issue again is usability. To our knowledge, no user-friendly graphical user interface exists.

The STATEMATE Verification Environment [20] is a tool to formally verify STATEMATE statecharts. The formal semantics of STATEMATE statecharts is described in [53]. To enable industrial usage, the tool hides the underlying formal verification technology from the user and supports the user by offering, among other features, a library of patterns for typical verification properties, a simulator and a waveform representation for counterexamples. The tool was used in several industrial projects in the avionics, automotive and railway industry to establish safety critical properties. We give some examples below.

An example from the avionics industry is described in [19]. A store management system of the British Aerospace is modeled using STATEMATE statecharts. The requirements are expressed as symbolic timing diagrams which provide a graphical interface to temporal logic. The STATEMATE Verification Environment gives a formal proof that the statecharts satisfy the requirements defined by the symbolic timing diagrams.

An example form the automotive industry is described in [18]. The controller of a brake management system for BMW is modeled with STATEMATE statecharts and activity charts. The requirements are formalized with symbolic timing diagrams. As in our models, to run the formal verification some abstractions are necessary, especially for large integers and real values.

An example from the railway industry is described in [54]. The radio-based signaling system is modeled with live sequence charts and activity charts. The requirements are captured with symbolic timing diagrams. Again, some variables need to be abstracted in order to be able to prove all properties.

To the best of our knowledge, the STATEMATE Verification Environment is not developed and not available anymore.

TOPCASED [72, 121] is an Eclipse-based toolkit. It includes a UML editor, and supports the mapping of requirements to UML diagrams and documentation generation. There exists a proof of concept implementation (AGATE) that translates from UML diagrams to the input language of the model checking tool UPPAAL [14]. Again this verification flow is not applicable for industrial use since it requires using three independent tools without an adequate Graphical User Interface (GUI), and the translation tool is not mature.

In more recent work Aoki and Matsuura [6] use UML activity diagrams and the model checker UPPAAL [14] to analyze the specification of a learning management system. Use cases are specified using activity diagrams and then translated to UPPAAL. The security requirements of an access control policy are specified in a table and then translated into formal properties. The approach probably also meets the Common Criteria assurance requirements for the Security Policy Model but the authors only describe the translation from the UML activity diagrams to the UPPAAL model based on the given example and it is not clear if a tool exists and at which level of maturity.

Unfortunately none of the above tools fits our purpose. We decided to use COSIDE as described in Section 3.5, which was used before to formally verify

the UML statecharts for programmable logic controllers [100].

3.7 Conclusions

Model checking is an efficient method for high Evaluation Assurance Level Common Criteria Security Policy Modeling. In Section 3.4.2 we show how the method we propose satisfies the Common Criteria assurance requirements for Security Policy Modeling. In Section 3.4.3 we describe a Security Policy Model of a Security IC that led to the first Common Criteria Evaluation Assurance Level 6 certificate in the German Common Criteria certification scheme. Our method is more efficient than previously proposed methods because model checking does not require any manual guidance to conduct proofs. In addition Security Policy Models made of UML statecharts can be understood and reviewed by engineers without formal methods background and the UML diagrams can be used as part of the specification. In Section 3.6.1 we give an overview of previously published work related to Security Policy Modeling and compare it to our method. To put it in a nutshell, our Security Policy Modeling method based on model checking is less time consuming, more user friendly, and still sufficient to uncover bugs in requirements (see Section 3.4.4).

The Security Policy Model required by Common Criteria proves that the functional specification satisfies security functional requirements. There is no direct link from the formally proven model to the implementation. We propose to generate test cases from the formal model and run them on the implementation to close this gap. First work on test case generation from a formal model for certification is published in [28]. A short discussion on related work can be found in Section 3.6.3.

In Section 3.5 we propose to integrate the Security Policy Modeling into the design process of a product by using UML statecharts for modeling. The idea to integrate the formal analysis of specifications and requirements into the design process by using a graphical statechart representation is not new. Several similar proposals have been made in the last 20 years, see Section 3.6.4. The most prominent example is the STATEMATE Verification Environment [20], which was used in safety critical areas such as the automotive, railway, or aerospace industries. In contrast we propose to use UML statecharts to verify security critical properties for banking and e-government products to satisfy the Common Criteria requirements for the Security Policy Model.

We think that a low initial barrier and good usability of the tools are important factors for the successful integration of formal verification into the design process. In our approach the security functional requirements have to be translated into LTL/CTL formula. This can be improvement, for example by providing templates for commonly used requirements or a pool of predefined properties. Other options could be an automatic translation from a predefined representation such as a table as defined in [6] into formal properties or the use of symbolic timing diagrams as used in [20].

As long as no easy integration into existing processes is possible formal analysis

of requirements will only be applied where required from external parties such as certification authorities. The additional costs of training engineers and developing tools, and the additional risks added to a project when changing the design process will not pay off in most projects. Only for complex, security or safety critical designs a formal model for the specification is clearly beneficial. In such cases the formal model can help to clarify the specification by removing ambiguities and uncovering inconsistencies or bugs early in the design process. Already the process of formalizing the specification helps to get a better understanding of the specification. In our experience problems in the specification are only confirmed by the formal proof, they are found in the formalization step.

Especially in combination with verification, a formal model of the specification can increase the quality and security of a product. For example, test cases can be generated from the model and run on the implementation. Linking the formal model to the implementation is an important step to increase the assurance that the implementation satisfies its requirements.

4
Conclusions

Linear Temporal Logic (LTL) was defined for specifying the behavior of reactive systems in the 1970s. At first only used in basic research it slowly found its way into industry. Nonetheless there are many open research topics both in the theoretical and the industrial area.

In the theoretical part of this thesis we work on synthesizing robust systems. Robustness is a property that is hard to specify with LTL but is obviously desirable. We show how robustness can be formally specified and verified and how a robust system can be synthesized from a robustness specification. We think that non-functional properties like robustness are a key factor for the success of LTL synthesis.

A lot of work remains to be done before LTL synthesis is applicable in an industrial use case. At the same time LTL verification in the form of model checking is already mature enough for industrial use.

In the industry related part of this work we introduce a new approach for formal security policy modeling. We use a model checker to show that the functional specification satisfies the security functional requirements. Two case studies demonstrate the industrial application of the new approach. Working on these two case studies showed that the main advantage of the formalization is to get a better and clearer understanding of the specification. Thus misunderstandings and contradictions in the specification can be removed before they are implemented.

Integrating formal verification into the design process of a smart card is still not state of the art although the increasing complexity and the need for security are a big challenge for commonly used design methodologies. Using Unified Modeling Language (UML) statecharts, a widely accepted design language, as input for the formal verification enables non-experts in formal methods to understand

the functional specification input to the formal verification tool without further training. In addition to having a formal proof, the formal model can help to improve the documentation and generate a common, unambiguous understanding of the specification. Such an approach contributes to a first time right product development and increases the assurance of security and correctness, enabling Common Criteria certifications with Evaluation Assurance Level 6. We think, the adaptation of formal modeling in the specification phase will largely depend on the usability of the tools and the possible effort and cost reduction in verification.

All in all LTL offers a wide range of beautiful research topics which will eventually raise our methods in design and verification to the next level of maturity.

A

Java Card Properties

This appendix shows the input file to the model checker for the Firewall Access Control Policy of a Java Card, see Section 3.4.4 for a description of the model.

Every line starting with // is a comment and will be ignored by the model checker.

Note that formulas of the form

```
LTLSPEC
  G(premises -> conclusion)
```

can be vacuously true if "premises" is false. Thus for every formula of that form we check that "premises" can evaluate to true with a formula of the form

```
CTLSPEC
  AG EF(premises).
```

```
1
2  INVAR
3  (inputJCREop = noop | bytecode = 0)
4
5  //--------------------------------------------------------
6
7  // FDP_ACF.1.2[FIREWALL]
8  // The TSF shall enforce the following rules to
       determine if an
9  // operation among controlled subjects and controlled
       objects is allowed:
10
```

```
11 // + R.JAVA.1
12 // S.PACKAGE may freely perform OP.ARRAY_ACCESS, OP.
       INSTANCE_FIELD, OP.INVK_VIRTUAL, OP.INVK_INTERFACE,
13 // OP.THROW or OP.TYPE_ACCESS upon any O.JAVAOBJECT whose
        Sharing attribute has value
14 // "JCRE entry point" or "global array".
15
16 // Show requirement for OP.INVK_VIRTUAL, OP.
       INVK_INTERFACE, OP.THROW, and OP.TYPE_ACCESS.
17
18 // The formula shows that the left hand side of the
       implication in the formula below can evaluate to true.
19 CTLSPEC
20 AG EF( CurrentlyActiveContext != 0 & (
       isJCREEntryPointObject | isGlobal) & ((bytecode >= 5)
       & (bytecode <= 7) | (bytecode >= 11) & (bytecode <=
       12)) & (state = idle))
21
22 LTLSPEC
23 G( CurrentlyActiveContext != 0 & (isJCREEntryPointObject
       | isGlobal) & ((bytecode >= 5) & (bytecode <= 7) | (
       bytecode >= 11) & (bytecode <= 12)) & (state = idle)
       -> X (state = idle))
24
25 // For OP.INSTANCE_FIELD the requirement can not be shown
       .
26 // getfield and putfield is not possible for JCRE entry
       point objects or global arrays.
27
28 // The formula shows that the left hand side of the
       implication in the formula below can evaluate to true.
29 CTLSPEC
30 AG EF( CurrentlyActiveContext != 0 & (
       isJCREEntryPointObject | isGlobal) & (bytecode = 3 |
       bytecode = 4) & (state = idle))
31
32 LTLSPEC
33 G( CurrentlyActiveContext != 0 & (isJCREEntryPointObject
       | isGlobal) & (bytecode = 3 | bytecode = 4) & (state =
        idle) -> X (state = locked))
34
35 // For OP.ARRAY_ACCESS the requirement can only be partly
        shown.
36
37 // The formula shows that the left hand side of the
```

```
    implication in the formula below can evaluate to true.
38  CTLSPEC
39  AG EF( CurrentlyActiveContext != 0 & isGlobal & (bytecode
        = 8 | bytecode = 9) & (state = idle))
40
41  LTLSPEC
42  G( CurrentlyActiveContext != 0 & isGlobal & (bytecode = 8
        | bytecode = 9) & (state = idle) -> X (state = idle))
43
44  // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
45  CTLSPEC
46  AG EF( CurrentlyActiveContext != 0 & isGlobal & bytecode
        = 13 & (state = idle))
47
48  // aastore of a global array is not possible
49  LTLSPEC
50  G( CurrentlyActiveContext != 0 & isGlobal & bytecode = 13
        & (state = idle) -> X (state = locked))
51
52  // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
53  CTLSPEC
54  AG EF( CurrentlyActiveContext != 0 &
        isJCREEntryPointObject & !isGlobal & (bytecode = 8 |
        bytecode = 9 | bytecode = 13) & (state = idle))
55
56  // The Java Card specification requires for astore and
        aload that the accessing context is either JCRE or the
         owner of the object if the object is not a global
        array.
57  // Only the JCRE can designate JCRE Entry Point Objects.
        Thus there are no JCRE Entry Point Objects that do not
         belong to the JCRE.
58  LTLSPEC
59  G( CurrentlyActiveContext != 0 & isJCREEntryPointObject &
         !isGlobal & (bytecode = 8 | bytecode = 9 | bytecode =
         13) & (state = idle) -> X (state = locked))
60  //--
61
62  // + R.JAVA.2
63  // S.PACKAGE may freely perform OP.ARRAY_ACCESS, OP.
        INSTANCE_FIELD, OP.INVK_VIRTUAL, OP.INVK_INTERFACE
64  // or OP.THROW upon any O.JAVAOBJECT whose Sharing
        attribute has value "Standard" and whose
```

```
65 // Lifetime attribute has value "PERSISTENT" only if O.
       JAVAOBJECT's Context attribute has the
66 // same value as the active context.
67
68 // The formula shows that the left hand side of the
       implication in the formula below can evaluate to true.
69 CTLSPEC
70 AG EF ( CurrentlyActiveContext != 0 & (bytecode >= 3 &
       bytecode <= 9 | bytecode = 13) & !(ValisGlobal |
       ValisJCREEntryPointObject |
       ValisShareableInterfaceObject) & (!
       ValisTransientObject | ValLifetimeTransient =
       clearonreset) & !(isGlobal | isJCREEntryPointObject |
       isShareableInterfaceObject) & (!isTransientObject |
       LifetimeTransient = clearonreset) &
       CurrentlyActiveContext = Owner & state = idle )
71
72 LTLSPEC
73 G ( CurrentlyActiveContext != 0 & (bytecode >= 3 &
       bytecode <= 9 | bytecode = 13) & !(ValisGlobal |
       ValisJCREEntryPointObject |
       ValisShareableInterfaceObject) & (!
       ValisTransientObject | ValLifetimeTransient =
       clearonreset) & !(isGlobal | isJCREEntryPointObject |
       isShareableInterfaceObject) & (!isTransientObject |
       LifetimeTransient = clearonreset) &
       CurrentlyActiveContext = Owner & state = idle -> X (
       state = idle))
74
75 // The formula shows that the left hand side of the
       implication in the formula below can evaluate to true.
76 CTLSPEC
77 AG EF ( CurrentlyActiveContext != 0 & (bytecode >= 3 &
       bytecode <= 9 | bytecode = 13) & !(ValisGlobal |
       ValisJCREEntryPointObject |
       ValisShareableInterfaceObject) & (!
       ValisTransientObject | ValLifetimeTransient =
       clearonreset) & !(isGlobal | isJCREEntryPointObject |
       isShareableInterfaceObject) & (!isTransientObject |
       LifetimeTransient = clearonreset) &
       CurrentlyActiveContext != Owner & state = idle )
78
79 // only if
80 LTLSPEC
81 G ( CurrentlyActiveContext != 0 & (bytecode >= 3 &
```

```
    bytecode <= 9 | bytecode = 13)& !(ValisGlobal |
    ValisJCREEntryPointObject |
    ValisShareableInterfaceObject)& (!ValisTransientObject
     | ValLifetimeTransient = clearonreset) & !(isGlobal |
     isJCREEntryPointObject | isShareableInterfaceObject)
    & (!isTransientObject | LifetimeTransient =
    clearonreset) & CurrentlyActiveContext != Owner &
    state = idle -> X (state = locked))
82  //--
83
84  // + R.JAVA.3
85  // S.PACKAGE may perform OP.TYPE_ACCESS upon an O.
        JAVAOBJECT whose Sharing attribute has value
86  // "SIO" only if O.JAVAOBJECT is being cast into (
        checkcast) or is being verified as being an
87  // instance of (instanceof) an interface that extends the
         Shareable interface.
88
89  // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
90  CTLSPEC
91  AG EF (CurrentlyActiveContext != 0 & (bytecode = 11 |
        bytecode = 12) & isShareableInterfaceObject &
        inputAccessShareableInterface & state = idle)
92
93  LTLSPEC
94  G (CurrentlyActiveContext != 0 & (bytecode = 11 |
        bytecode = 12) & isShareableInterfaceObject &
        inputAccessShareableInterface & state = idle -> X (
        state = idle))
95
96  // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
97  CTLSPEC
98  AG EF (CurrentlyActiveContext != 0 & Owner !=
        CurrentlyActiveContext & (bytecode = 11 | bytecode =
        12) & isShareableInterfaceObject & !
        inputAccessShareableInterface & state = idle)
99
100 //only if holds when the currently active context is not
        the Owner
101 LTLSPEC
102 G (CurrentlyActiveContext != 0 & Owner !=
        CurrentlyActiveContext & (bytecode = 11 | bytecode =
        12) & isShareableInterfaceObject & !
```

```
    inputAccessShareableInterface & state = idle -> X (
    state = locked))
103 //--
104
105 // + R.JAVA.4
106 // S.PACKAGE may perform OP.INVK_INTERFACE upon an O.
    JAVAOBJECT whose Sharing attribute has the value
107 // "SIO", and whose Context attribute has the value "
    Package AID", only if the invoked interface
108 // method extends the Shareable interface and one of the
    following conditions applies:
109 // a) The value of the attribute Selection Status of the
    package whose AID is "Package AID" is "Multiselectable
    ",
110 // b) The value of the attribute Selection Status of the
    package whose AID is "Package AID" is "Non-
    multiselectable", and either "Package AID" is the
    value of the currently selected applet or otherwise "
    Package AID" does not occur in the attribute Active
    Applets.
111
112 // + R.JAVA.4 (a)
113 // The formula shows that the left hand side of the
    implication in the formula below can evaluate to true.
114 CTLSPEC
115 AG EF (CurrentlyActiveContext != 0 & bytecode = 6 &
    isShareableInterfaceObject &
    inputAccessShareableInterface & inputLCSelectionStatus
     = multiselectable & state = idle)
116
117 LTLSPEC
118 G (CurrentlyActiveContext != 0 & bytecode = 6 &
    isShareableInterfaceObject &
    inputAccessShareableInterface & inputLCSelectionStatus
     = multiselectable & state = idle -> X (state = idle))
119
120 // The formula shows that the left hand side of the
    implication in the formula below can evaluate to true.
121 CTLSPEC
122 AG EF (CurrentlyActiveContext != 0 & Owner !=
    CurrentlyActiveContext & bytecode = 6 &
    isShareableInterfaceObject & !
    inputAccessShareableInterface & state = idle)
123
124 //only if holds when the currently active context is not
```

```
    the Owner
125 LTLSPEC
126 G (CurrentlyActiveContext != 0 & Owner !=
    CurrentlyActiveContext & bytecode = 6 &
    isShareableInterfaceObject & !
    inputAccessShareableInterface & state = idle -> X (
    state = locked))
127
128 // + R.JAVA.4 (b)
129 // The formula shows that the left hand side of the
    implication in the formula below can evaluate to true.
130 CTLSPEC
131 AG EF (CurrentlyActiveContext != 0 & bytecode = 6 &
    isShareableInterfaceObject &
    inputAccessShareableInterface & inputLCSelectionStatus
     = nonmultiselectable & (inputInCtxOfCurSelApplet | !
    inputActiveOnOtherLC) & state = idle)
132
133 LTLSPEC
134 G (CurrentlyActiveContext != 0 & bytecode = 6 &
    isShareableInterfaceObject &
    inputAccessShareableInterface & inputLCSelectionStatus
     = nonmultiselectable & (inputInCtxOfCurSelApplet | !
    inputActiveOnOtherLC) & state = idle -> X (state =
    idle))
135
136 // The formula shows that the left hand side of the
    implication in the formula below can evaluate to true.
137 CTLSPEC
138 AG EF (CurrentlyActiveContext != 0 & Owner !=
    CurrentlyActiveContext & bytecode = 6 &
    isShareableInterfaceObject &
    inputAccessShareableInterface & inputLCSelectionStatus
     = nonmultiselectable & !inputInCtxOfCurSelApplet &
    inputActiveOnOtherLC & state = idle)
139
140 //only if holds when the currently active context is not
    the Owner
141 LTLSPEC
142 G (CurrentlyActiveContext != 0 & Owner !=
    CurrentlyActiveContext & bytecode = 6 &
    isShareableInterfaceObject &
    inputAccessShareableInterface & inputLCSelectionStatus
     = nonmultiselectable & !inputInCtxOfCurSelApplet &
    inputActiveOnOtherLC & state = idle -> X (state =
```

```
    locked))
143 //--
144
145 // + R.JAVA.5
146 // S.PACKAGE may perform OP.CREATE only if the value of
    the Sharing parameter is "Standard".
147
148 // S.PACKAGE may perform OP.CREATE if the value of the
    Sharing parameter is "Standard".
149 CTLSPEC
150 EF( CurrentlyActiveContext != 0 & inputJCREop =
    createobject & inputSharing = standard & state = idle
    & EX (state = idle))
151
152 // The formula shows that the left hand side of the
    implication in the formula below can evaluate to true.
153 CTLSPEC
154 AG EF (CurrentlyActiveContext != 0 & inputJCREop =
    createobject & inputSharing = standard & (
    inputLifetimeTransient = clearonreset |
    CurrentlyActiveContext = SelectedAppletContext) &
    state = idle)
155
156 // S.PACKAGE may always perform OP.CREATE if the value of
     the Sharing parameter is "Standard" and
157 // the lifetime is CLEAR_ON_RESET or the currently active
     context is the selected applet context.
158 LTLSPEC
159 G( CurrentlyActiveContext != 0 & inputJCREop =
    createobject & inputSharing = standard & (
    inputLifetimeTransient = clearonreset |
    CurrentlyActiveContext = SelectedAppletContext) &
    state = idle -> X (state = idle))
160
161 // The formula shows that the left hand side of the
    implication in the formula below can evaluate to true.
162 CTLSPEC
163 AG EF (CurrentlyActiveContext != 0 & inputJCREop =
    createobject & state = idle & EX(state = idle))
164
165 //The formula below is false.
166 //The only if direction can not be proven because S.
    PACKAGE is allowed to e.g. create a SIO.
167 LTLSPEC
168 G( CurrentlyActiveContext != 0 & inputJCREop =
```

```
    createobject & state = idle & X(state = idle) -> (
    inputSharing = standard))
169
170 // The PP also states the following:
171 // Footnote for OP.CREATE: This rule enforces that
    shareable transient objects are not allowed.
172 LTLSPEC
173 G(isShareableInterfaceObject -> !isTransientObject)
174 //--
175
176
177 // FDP_ACF.1.3[FIREWALL]
178 // The TSF shall explicitly authorise access of subjects
    to objects based on the following additional rules:
179 // 1) The subject S.JCRE can freely perform OP.JAVA() and
     OP.CREATE, with the exception given in
180 //    FDP_ACF.1.4/FIREWALL, provided it is the Currently
    Active Context.
181
182 // The formula shows that the left hand side of the
    implication in the formula below can evaluate to true.
183 CTLSPEC
184 AG EF (CurrentlyActiveContext = 0 & (bytecode >= 3 &
    bytecode <= 9 | bytecode >= 11 & bytecode <= 13) & (
    LifetimeTransient != clearondeselect | !
    isTransientObject | Owner = SelectedAppletContext &
    SelectedAppletContext = CurrentlyActiveContext) &
    state = idle)
185
186 // Additionally for transient CLEAR_ON_DESELECT objects
    the selected applet context has to be the currently
    active context (JCRE Specification 6.5.1).
187 LTLSPEC
188 G(CurrentlyActiveContext = 0 & (bytecode >= 3 & bytecode
    <= 9 | bytecode >= 11 & bytecode <= 13) & (
    LifetimeTransient != clearondeselect | !
    isTransientObject | Owner = SelectedAppletContext &
    SelectedAppletContext = CurrentlyActiveContext) &
    state = idle  -> X (state = idle))
189
190 // The formula shows that the left hand side of the
    implication in the formula below can evaluate to true.
191 CTLSPEC
192 AG EF (CurrentlyActiveContext = 0 & inputJCREop =
    createobject & (inputLifetimeTransient !=
```

```
    clearondeselect | !inputIsTransientObject |
    CurrentlyActiveContext = SelectedAppletContext) &
    inputSharing != sio & state = idle)
193
194 // Additionally the JCRE can not create sharable
    interface objects.
195 LTLSPEC
196 G(CurrentlyActiveContext = 0 & inputJCREop = createobject
     & (inputLifetimeTransient != clearondeselect | !
    inputIsTransientObject | CurrentlyActiveContext =
    SelectedAppletContext) & inputSharing != sio & state =
     idle  -> X (state = idle))
197 //--
198
199 // 2) The only means that the subject S.JCVM shall
    provide for an application to execute native code is
200 //    the invocation of a Java Card API method (through
    OP.INVK_INTERFACE or OP.INVK_VIRTUAL).
201
202 // Native code is not modeled because it is not part of
    the JCRE specification regarding the firewall.
203 //--
204
205 // FDP_ACF.1.4[FIREWALL]
206 // The TSF shall explicitly deny access of subjects to
    objects based on the following additional rules:
207 // 1) Any subject with OP.JAVA upon an O.JAVAOBJECT whose
     LifeTime attribute has value "CLEAR_ON_DESELECT" if
208 //    O.JAVAOBJECT's Context attribute is not the same as
     the Selected Applet Context.
209
210 // The formula shows that the left hand side of the
    implication in the formula below can evaluate to true.
211 CTLSPEC
212 AG EF ((bytecode >= 3 & bytecode <= 9 | bytecode >= 11 &
    bytecode <= 13) & LifetimeTransient = clearondeselect
    & isTransientObject & Owner != SelectedAppletContext &
     state = idle )
213
214 LTLSPEC
215 G ((bytecode >= 3 & bytecode <= 9 | bytecode >= 11 &
    bytecode <= 13) & LifetimeTransient = clearondeselect
    & isTransientObject & Owner != SelectedAppletContext &
     state = idle -> X(state = locked))
216 //--
```

```
217
218 // 2) Any subject attempting to create an object by the
        means of OP.CREATE and a "CLEAR_ON_DESELECT" LifeTime
        parameter
219 //    if the active context is not the same as the
        Selected Applet Context.
220
221 // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
222 CTLSPEC
223 AG EF (inputJCREop = createobject &
        inputLifetimeTransient = clearondeselect &
        inputIsTransientObject & CurrentlyActiveContext !=
        SelectedAppletContext & state = idle)
224
225 LTLSPEC
226 G(inputJCREop = createobject & inputLifetimeTransient =
        clearondeselect & inputIsTransientObject &
        CurrentlyActiveContext != SelectedAppletContext &
        state = idle -> X (state = locked))
227
228 //--------------------------------------------------------
229
230 // FDP_ROL.1.1[FIREWALL]
231 // The TSF shall enforce the FIREWALL access control SFP
        and the JCVM information flow
232 // control SFP to permit the rollback of the operations
        OP.JAVA and OP.CREATE on the object O.JAVAOBJECT.
233
234 // Rollback is not modeled since restore is either
        performed before the VM is started or
235 // an invokestatic occurs (which runs outside of the
        Firewall).
236 //--
237
238 // FDP_ROL.1.2[FIREWALL]
239 // The TSF shall permit operations to be rolled back
        within the scope of a select(), deselect(),
240 // process(), install() or uninstall() call,
        notwithstanding the restrictions given in [28], 7.7,
241 // within the bounds of the Commit Capacity ([28], 7.8),
        and those described in [18].
242
243 // Rollback is not modeled since restore is either
        performed before the VM is started or
```

```
244 // an invokestatic occurs (which runs outside of the
        Firewall).
245
246 //--------------------------------------------------------
247
248 // FIA_UID.2.1[AID]
249 // The TSF shall require each user to be successfully
        identified before allowing any other TSF mediated
250 // actions on behalf of that user.
251
252 // The currently active context is always 0 (S.JCRE), or
        1 or 2 (S.PACKAGE).
253 LTLSPEC
254 G((CurrentlyActiveContext >= 0) & (CurrentlyActiveContext
        <= 2))
255
256 //--------------------------------------------------------
257
258 // FMT_MSA.1.1[JCRE]
259 // The TSF shall enforce the FIREWALL access control SFP
        to restrict the ability to modify the security
260 // attributes Selected Applet Context to the Java Card RE
        .
261
262 // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
263 CTLSPEC
264 AG EF (inputJCREop = selectedappletcontextswitch &
        CurrentlyActiveContext != 0 & state = idle)
265
266 // A request to change the selected applet context that
        is not issued by the JCRE leads to the locked state.
267 LTLSPEC
268 G (inputJCREop = selectedappletcontextswitch &
        CurrentlyActiveContext != 0 & state = idle -> X (state
         = locked))
269
270 // The JCRE is able to switch the selected applet context
        .
271 CTLSPEC
272 AG EF (inputJCREop = selectedappletcontextswitch &
        CurrentlyActiveContext = 0 & state = idle -> EX(state
        = idle))
273
274 //--------------------------------------------------------
```

```
275
276 // FMT_MSA.1.1[JCVM]
277 // The TSF shall enforce the FIREWALL access control SFP
        and the JCVM information flow
278 // control SFP to restrict the ability to modify the
        security attributes Currently Active Context
279 // and Active Applets to the Java Card VM (S.JCVM).
280
281 // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
282 CTLSPEC
283 AG EF (inputJCREop = currentlyactivecontextswitch & !
        inputJCVMisRequester & state = idle)
284
285 // A request to change the currently active context that
        is not issued by the JCVM leads to the locked state.
286 LTLSPEC
287 G (inputJCREop = currentlyactivecontextswitch & !
        inputJCVMisRequester & state = idle -> X (state =
        locked))
288
289
290 // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
291 CTLSPEC
292 AG EF (inputJCREop = currentlyactivecontextswitch & !
        inputJCVMisRequester & CurrentlyActiveContext = 0 &
        state = idle)
293
294 CTLSPEC
295 AG EF (inputJCREop = currentlyactivecontextswitch & !
        inputJCVMisRequester & CurrentlyActiveContext = 1 &
        state = idle)
296
297 CTLSPEC
298 AG EF (inputJCREop = currentlyactivecontextswitch & !
        inputJCVMisRequester & CurrentlyActiveContext = 2 &
        state = idle)
299
300 // If the requester is not the JCVM, the currently active
         context does not change.
301 LTLSPEC
302 G (inputJCREop = currentlyactivecontextswitch & !
        inputJCVMisRequester & CurrentlyActiveContext = 0 &
        state = idle -> X (state = locked &
```

```
    CurrentlyActiveContext = 0))
303
304 LTLSPEC
305 G (inputJCREop = currentlyactivecontextswitch & !
    inputJCVMisRequester & CurrentlyActiveContext = 1 &
    state = idle -> X (state = locked &
    CurrentlyActiveContext = 1))
306
307 LTLSPEC
308 G (inputJCREop = currentlyactivecontextswitch & !
    inputJCVMisRequester & CurrentlyActiveContext = 2 &
    state = idle -> X (state = locked &
    CurrentlyActiveContext = 2))
309
310
311 // If the requester is the JCVM, the currently active
    context can be changed.
312 CTLSPEC
313 AG EF (inputJCREop = currentlyactivecontextswitch &
    inputJCVMisRequester & CurrentlyActiveContext = 0 &
    state = idle -> EX(state = idle &
    CurrentlyActiveContext != 0))
314
315 CTLSPEC
316 AG EF (inputJCREop = currentlyactivecontextswitch &
    inputJCVMisRequester & CurrentlyActiveContext = 1 &
    state = idle -> EX(state = idle &
    CurrentlyActiveContext != 1))
317
318 CTLSPEC
319 AG EF (inputJCREop = currentlyactivecontextswitch &
    inputJCVMisRequester & CurrentlyActiveContext = 2 &
    state = idle -> EX(state = idle &
    CurrentlyActiveContext != 2))
320
321 // The list of active applets is not modeled, we only
    model if an object is owned by an applet that
322 // is active on another logical channel using a
    nondeterministic input variable (inputActiveOnOtherLC)
    .
323 // Nondeterministic input variables can change at every
    time step.
324
325 //--------------------------------------------------------
326
```

```
327 // FMT_MSA.2.1[FIREWALL_JCVM]
328 // The TSF shall ensure that only secure values are
        accepted for all the security attributes
329 // of subjects and objects defined in the FIREWALL access
        control SFP and the
330 // JCVM information flow control SFP.
331 // -> see AppNote
332
333 // The Context attribute of an O.JAVAOBJECT must
        correspond to that of an installed
334 // applet or be "Java Card RE".
335 LTLSPEC
336 G (Owner = 0 | (Owner = 1 & Applet1installed) | (Owner =
        2 & Applet2installed))
337
338 // An O.JAVAOBJECT whose Sharing attribute is a Java Card
        RE entry point or a global
339 // array necessarily has "Java Card RE" as the value for
        its Context security attribute.
340
341 // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
342 CTLSPEC
343 AG (EF (isJCREEntryPointObject | isGlobal))
344
345 LTLSPEC
346 G ((isJCREEntryPointObject | isGlobal) -> (Owner = 0))
347
348 // An O.JAVAOBJECT whose Sharing attribute value is a
        global array necessarily has
349 // "array of primitive type" as a JavaCardClass security
        attribute's value.
350
351 // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
352 CTLSPEC
353 AG EF (isGlobal)
354
355 LTLSPEC
356 G (isGlobal -> isArray)
357
358 // Any O.JAVAOBJECT whose Sharing attribute value is not
        "Standard" has a
359 // PERSISTENT-LifeTime attribute's value.
360
```

```
361 // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
362 CTLSPEC
363 AG EF (isJCREEntryPointObject | isGlobal |
        isShareableInterfaceObject)
364
365 // The PP states the following:
366 // Footnote of Lifetime: Transient objects of type
        CLEAR_ON_RESET behave like persistent objects in that
367 // they can be accessed only when the Currently Active
        Context is the object's context.
368 LTLSPEC
369 G ((isJCREEntryPointObject | isGlobal |
        isShareableInterfaceObject)-> (!isTransientObject |
        LifetimeTransient = clearonreset))
370
371 // Any O.JAVAOBJECT whose LifeTime attribute value is not
        PERSISTENT has an array
372 // type as JavaCardClass attribute's value.
373
374 // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
375 CTLSPEC
376 AG EF (isTransientObject & LifetimeTransient =
        clearondeselect)
377
378 LTLSPEC
379 G (isTransientObject & LifetimeTransient =
        clearondeselect -> isArray)
380
381 //--------------------------------------------------------
382
383 // FMT_MSA.3.1[FIREWALL]
384 // The TSF shall enforce the FIREWALL access control SFP
        to provide restrictive default
385 // values for security attributes that are used to
        enforce the SFP.
386 // see App-Notes
387
388 // The formula shows that the left hand side of the
        implication in the formula below can evaluate to true.
389 CTLSPEC
390 AG EF(inputJCREop = createobject & state = idle & EX(
        state = idle))
391
```

```
392 LTLSPEC
393 G (inputJCREop = createobject & state = idle & X(state =
        idle) -> X(Owner = CurrentlyActiveContext))
394 //--
395
396 // FMT_MSA.3.2[FIREWALLEditoriallyRefined]
397 // The TSF shall not allow any role to specify
        alternative initial values to override
398 // the default values when an object or information is
        created.
399
400 // The formula is false, because there exists no path
        such that an object is created with a Context that
401 // is not the currently active context.
402 CTLSPEC
403 EF(inputJCREop = createobject & state = idle & EX(state =
         idle) & EX(Owner != CurrentlyActiveContext))
404
405 //-------------------------------------------------------
406
407 // FMT_SMF.1.1
408 // The TSF shall be capable of performing the following
        management functions:
409 // * modify the Currently Active Context, the Selected
        Applet Context and the Active Applets
410
411 CTLSPEC
412 AG EF( CurrentlyActiveContext = 0 & EX(
        CurrentlyActiveContext != 0))
413
414 CTLSPEC
415 AG EF( CurrentlyActiveContext != 0 & EX(
        CurrentlyActiveContext = 0))
416
417 CTLSPEC
418 AG EF( SelectedAppletContext = 0 & EX(
        SelectedAppletContext != 0))
419
420 CTLSPEC
421 AG EF( SelectedAppletContext != 0 & EX(
        SelectedAppletContext = 0))
422
423 // The list of active applets is not modeled, we only
        model if an object is owned by an applet that
424 // is active on another logical channel using a
```

```
    nondeterministic input variable (inputActiveOnOtherLC)
    .
425 // Nondeterministic input variables can change at every
    time step.
426 CTLSPEC
427 AG EF (inputActiveOnOtherLC & EX (!inputActiveOnOtherLC))
428
429 CTLSPEC
430 AG EF (!inputActiveOnOtherLC & EX (inputActiveOnOtherLC))
431
432 //-----------------------------------------------------------
433
434 // FMT_SMR.1.1
435 // The TSF shall maintain the roles:
436 // * Java Card RE (JCRE),
437 // * Java Card VM (JCVM).
438
439 CTLSPEC
440 EF (CurrentlyActiveContext = 0)
441
442 // The JCVM is modeled with a nondeterministic input
    variable that can change at every time step.
443 CTLSPEC
444 EF (inputJCVMisRequester)
445 //--
446
447 FMT_SMR.1.2
448 // The TSF shall be able to associate users with roles.
449
450 // Package AIDs are assigned, the JCRE has context 0.
```

Bibliography

[1] Agence nationale de la sécurité des systèmes d'information. *Rapport de certification ANSSI-CC-2009/50 Microcontrôleurs sécurisés ST23YR48A et ST23YR80A*, 2009.

[2] *Application Notes and Interpretation of the Scheme (AIS 34), Evaluation Methodology for CC Assurance Classes for EAL5+ (CC v2.3 & v3.1) and EAL6 (CC v3.1)*, September 2009.

[3] *Application Notes and Interpretation of the Scheme (AIS 39), Formal Methods - Guideline for the Development and Evaluation of the formal security policy models in the scope of ITSEC and Common Criteria*, September 2009.

[4] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, Oct. 1985.

[5] R. Alur, A. Kanade, and G. Weiss. Ranking automata and games for prioritized requirements. In *Computer Aided Verification*, pages 240–253, 2008.

[6] Y. Aoki and S. Matsuura. Verifying security requirements using model checking technique for uml-based requirements specification. In *1st IEEE International Workshop on Requirements Engineering and Testing, RET 2014, Karlskrona, Sweden, August 26, 2014*, pages 18–25, 2014.

[7] M. Archer. Tame: Using pvs strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 2000.

[8] A. Arora. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transatcions of Software Engineering*, 19:1015–1027, 1993.

[9] P. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26:125–185, 2004.

[10] C. Baier and J. Katoen. *Principles of model checking*. The MIT Press, 2008.

[11] G. Barthe and G. Dufay. Formal methods for smartcard security. In *Foundations of Security Analysis and Design III, FOSAD 2004/2005 Tutorial Lectures*, pages 133–177, 2005.

[12] B. Beckert, D. Bruns, and S. Grebing. Mind the gap: Formal verification and the common criteria (discussion paper). In *6th International Verification Workshop, VERIFY-2010, Edinburgh, UK, July 20-21, 2010*, pages 4–12, 2010.

[13] S. Z. Béguelin. Formalisation and verification of the globalplatform card specification using the B method. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Second International Workshop, CASSIS 2005, Nice, France, March 8-11, 2005, Revised Selected Papers*, pages 155–173, 2005.

[14] G. Behrmann, A. David, K. G. Larsen, O. Moller, P. Pettersson, and W. Yi. Uppaal – present and future. In *Proc. 40th IEEE Conf. Decision and Control*, pages 2881–2886, 2001.

[15] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. In *Proceedings of the Eight ACM Symposium on Principles of Programming Languages*, pages 164–176, 1981.

[16] G. Beuster and K. Greimel. Formal security policy models for smart card evaluations. In *International Symposium on Applied Computing (SAC)*, 2012.

[17] J. Bicarregui, J. S. Fitzgerald, P. G. Larsen, and J. C. P. Woodcock. Industrial practice in formal methods: A review. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 810–813, 2009.

[18] T. Bienmüller, J. Bohn, H. Brinkmann, U. Brockmeyer, W. Damm, H. Hungar, and P. Jansen. Verification of automotive control units. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel)*, 1999.

[19] T. Bienmüller, U. Brockmeyer, W. Damm, G. Döhmen, C. Eßmann, H. Holberg, H. Hungar, B. Josko, R. Schlör, G. Wittich, H. Wittke, G. Clements, J. Rowlands, and E. Sefton. Formal verification of an avionics application using abstraction and symbolic model checking. In *Towards System Safety – Proc. 7th Safety-critical Systems Symposium*, pages 150–173, 1999.

[20] T. Bienmüller, W. Damm, and H. Wittke. The Statemate verification environment. In *Proc. Int. Conf. Computer-Aided Verification*, volume 1855 of *LNCS*, pages 561–567, 2000.

[21] R. Bloem, K. Chatterjee, K. Greimel, T. Henzinger, G. Hofferek, B. Jobstmann, B. Könighofer, and R. Könighofer. Synthesizing robust systems. *Acta Inf.*, 51(3-4):193–220, 2014.

[22] R. Bloem, K. Chatterjee, K. Greimel, T. Henzinger, and B. Jobstmann. Robustness in the presence of liveness. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 410–424, 2010.

[23] R. Bloem, K. Chatterjee, T. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *Int. Conf. Computer Aided Verification (CAV)*, pages 140–156, 2009.

[24] R. Bloem, R. Ehlers, S. Jacobs, and R. Könighofer. How to handle assumptions in synthesis. In *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.*, pages 34–50, 2014.

[25] R. Bloem, R. Ehlers, and R. Könighofer. Cooperative reactive synthesis. In *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*, pages 394–410, 2015.

[26] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *In Proceedings of the Design, Automation and Test in Europe*, pages 1188–1193, 2007.

[27] R. Bloem, K. Greimel, T. Henzinger, and B. Jobstmann. Synthesizing robust systems. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 85–92, 2009.

[28] R. Bloem, K. Greimel, R. Könighofer, and F. Roeck. Model-based MCDC testing of complex decisions for the java card applet firewall. In *The Fifth International Conference on Advances in System Testing and Validation Lifecycle VALID 2013, Venice, Italy, October 27 - November 1, 2013*, pages 1–7, 2013.

[29] R. Bloem, D. M. Hein, F. Roeck, and R. Schumi. Case study: Automatic test case generation for a secure cache implementation. In *Tests and Proofs - 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22-24, 2015. Proceedings*, pages 58–75, 2015.

[30] M. Broy, W. Damm, S. Henkler, K. Pohl, A. Vogelsang, and T. Weyer. Introduction to the spes modeling framework. In K. Pohl, H. Hnninger, R. Achatz, and M. Broy, editors, *Model-Based Engineering of Embedded Systems*, pages 31–49. Springer Berlin Heidelberg, 2012.

[31] A. D. Brucker, A. Feliachi, Y. Nemouchi, and B. Wolff. Test program generation for a microprocessor - A case-study. In *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, pages 76–95, 2013.

[32] *Security IC Platform Protection Profile with Augmentation Packages Version 1.0*, 2014.

[33] *Security IC Platform Protection Profile Version 1.0*, June 2007.

[34] A. Campetelli, M. Junker, B. Böhm, M. Davidich, V. Koutsoumpas, X. Zhu, and J. Wehrstedt. A model-based approach to formal verification in early development phases: A desalination plant case study. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.-18. März 2015.*, pages 91–100, 2015.

[35] *Common Criteria for Information Technology Security Evaluation Version 3.1 Revision 4, Part 1: Introduction and general model*, September 2012.

[36] *Common Criteria for Information Technology Security Evaluation Version 3.1 Revision 4*, September 2012.

[37] *Common Criteria for Information Technology Security Evaluation Version 3.1 Revision 4, Part 2: Security functional components*, September 2012.

[38] *Common Criteria for Information Technology Security Evaluation Version 3.1 Revision 4, Part 3: Security assurance components*, September 2012.

[39] *Common Methodology for Information Technology Security Evaluation Version 3.1 Revision 4, Evaluation methodology*, September 2012.

[40] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[41] K. Chatterjee, T. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *International Conference on Concurrency Theory (CONCUR)*, pages 147–161, 2008.

[42] K. Chatterjee, T. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. *J. ACM*, 62(1):9:1–9:34, 2015.

[43] K. Chatterjee, T. Henzinger, and N. Piterman. Generalized parity games. In *10th International Conference on Foundations of Software Science and Computation Structures*, pages 153–167. Springer, 2007. LNCS 4423.

[44] C. Cheng, H. Rueß, A. Knoll, and C. Buckl. Synthesis of fault-tolerant embedded systems using games: From theory to practice. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 118–133, 2011.

[45] B. Chetali. Security testing and formal methods for high levels certification of smart cards. In *Tests and Proofs, Third International Conference, TAP 2009, Zurich, Switzerland, July 2-3, 2009. Proceedings*, pages 1–5, 2009.

[46] B. Chetali and Q. Nguyen. Industrial use of formal methods for a high-level security evaluation. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, pages 198–213, 2008.

[47] A. Church. Logic, arithmetic and automata. In *Proceedings International Mathematical Congress*, 1962.

[48] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 359–364, 2002.

[49] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, pages 495–499, 1999.

[50] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[51] J. E. R. Cury and B. H. Krogh. Robustness of supervisors for discrete-event systems. *IEEE Transactions on Automatic Control*, 44(2):376–379, 1999.

[52] W. Damm and B. Finkbeiner. Does it pay to extend the perimeter of a world model? In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 12–26, 2011.

[53] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of statemate designs. In *Compositionality: The Significant Difference*, pages 186–238. Springer-Verlag, 1998.

[54] W. Damm and J. Klose. Verification of a radio-based signaling system using the STATEMATE verification environment. *Formal Methods in System Design*, 19(2):121–141, 2001.

[55] A. Dasdan, S. S. Irani, and R. K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Proceedings of the Design Automation Conference*, pages 37–42, 1999.

[56] A. M. Davis. *Software Requirements — Analysis and Specification*. Prentice Hall, 1990.

[57] L. de Alfaro and M. Faella. Accelerated algorithms for 3-color parity games with an application to timed games. In *Nineteenth International Conference on Computer Aided Verification (CAV'07)*, pages 108–120, Berlin, 2007. Springer-Verlag. LNCS 4590.

[58] E. Dijkstra. Cooperating sequential processes. In Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[59] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[60] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.*, 22(1):9, 2013.

[61] N. D'Ippolito, V. Braberman, D. Sykes, and S. Uchitel. Robust degradation and enhancement of robot mission behaviour in unpredictable environments. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering, CTSE@SIGSOFT FSE 2015, Bergamo, Italy, August 31 - September 04, 2015*, pages 26–33, 2015.

[62] L. Doyen, T. Henzinger, A. Legay, and D. Nickovic. Robustness of sequential circuits. In *10th International Conference on Application of Concurrency to System Design, ACSD 2010, Braga, Portugal, 21-25 June 2010*, pages 77–84, 2010.

[63] D. D'Souza and M. Gopinathan. Conflict-tolerant features. In *Computer Aided Verification (CAV)*, pages 227–239, 2008.

[64] A. Ebnenasir, S. S. Kulkarni, and A. Arora. Ftsyn: a framework for automatic synthesis of fault-tolerance. *Software Tools for Technology Transfer*, 10:455–471, 2008.

[65] R. Ehlers. Generalized rabin(1) synthesis with applications to robust system synthesis. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 101–115, 2011.

[66] R. Ehlers and U. Topcu. Resilience to intermittent assumption violations in reactive synthesis. In *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014*, pages 203–212, 2014.

[67] C. Eisner. Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. In *CHARME*, pages 97–109, 1999.

[68] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pages 368–377, Oct. 1991.

[69] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.

[70] EMV. *Integrated Circuit Card Specifications for Payment Systems, Book 3, Application Specification, Version 4.3*, November 2011.

[71] M. Faella. Games you cannot win. In *Workshop on Games and Automata for Synthesis and Validation*, 2007.

[72] P. Farail, P. Goutillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Cregut, and M. Pantel. The TOPCASED project: a toolkit in open source for critical aeronautic systems design. In *Proc. Embedded Real Time Software and Systems*, 2006.

[73] FBK-irst. *NuSMV 2.5 User Manual*.

[74] G. Fey and R. Drechsler. A basis for formal robustness checking. In *ISQED*, pages 784–789, 2008.

[75] Fraunhofer. *IDE:StateCharts for Formal verification*.

[76] Fraunhofer IIS/EAS. *Coside - SystemC AMS Design Environment*, 2012. available online at `http://www.eas.iis.fraunhofer.de/en/business_areas/microelectronic_systems/system_development/coside.html`.

[77] A. Fuxman, J. Mylopoulos, M. Pistore, and P. Traverso. Model checking early requirements specifications in tropos. In *5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada*, pages 174–181, 2001.

[78] gemalto. *Formal Assurance on the JavaCard Virtual Machine embedded in Usimera Protect, Security Target*, September 2007.

[79] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

[80] H. Gimbert and W. Zielonka. Games where you can play optimally without any memory. In *CONCUR*, pages 428–442, 2005.

[81] A. Girault and É. Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Formal Methods in System Design*, 35(2):190–225, 2009.

[82] GlobalPlatform. *GlobalPlatform Card Specification, Version 2.2.1*, January 2011.

[83] K. Greimel, R. Bloem, B. Jobstmann, and M. Vardi. Open implication. In *Proc. Int. Colloquium on Automata, Languages and Programming (ICALP'08)*, pages 361–372, 2008. LNCS 5126.

[84] K. Greimel, N. Seßler, and T. Klotz. Model checking specifications of smart cards. In *Annual Conference of the IEEE Industrial Electronics Society (IECON)*, 2013.

[85] D. Greve, M. Wilding, R. Richards, and W. Vanfleet. Formalizing security policies for dynamic and distributed systems. In *Systems and Software Technology Conference (SSTC 2005)*, 2005.

[86] I. Grobelna, M. Grobelny, and M. Adamski. Model checking of uml activity diagrams in logic controllers design. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems*, DepCoS-RELCOMEX, 2014.

[87] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002.

[88] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, 2006.

[89] J. Helbig, R. Schlör, W. Damm, G. Döhmen, and P. Kelb. VHDL/S - integrating statecharts, timing diagrams, and VHDL. *Microprocessing and Microprogramming*, 38(1-5):571–580, 1993.

[90] T. Henzinger. Two challenges in embedded systems design: Predictability and robustness. *Philosophical Transactions of the Royal Society*, 2008.

[91] T. Henzinger, J. Otop, and R. Samanta. Lipschitz robustness of finite-state transducers. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, pages 431–443, 2014.

[92] T. Henzinger, J. Otop, and R. Samanta. Lipschitz robustness of timed I/O systems. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 250–267, 2016.

[93] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[94] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 117–124, 2006.

[95] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification*, pages 258–262, 2007.

[96] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *Journal of Computer and System Sciences*, 78(2):441460, 2012.

[97] M. Jurdziński. Small progress measures for solving parity games. In *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301, Lille, France, Feb. 2000. Springer. LNCS 1770.

[98] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[99] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Comput. Sci.*, 298:583–626, 2003.

[100] T. Klotz, E. Fordran, B. Straube, and J. Haufe. Formal verification of UML-modeled machine controls. In *Proc. IEEE Conf. Emerging Technologies and Factory Automation*, pages 1–7, 2009.

[101] S. S. Kulkarni and A. Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2:1–15, 2005.

[102] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Courier Dover Publications, 1976.

[103] J. Lilius and I. Paltor. vUML: A tool for verifying UML models. In *Proc. Int. Conf. Automated Software Engineering*, pages 255–258, 1999.

[104] K. Loer and M. D. Harrison. Towards usable and relevant model checking techniques for the analysis of dependable interactive systems. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*, pages 223–226, 2002.

[105] R. Majumdar, E. Render, and P. Tabuada. Robust discrete synthesis against unspecified disturbances. In *Proceedings of the 14th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2011, Chicago, IL, USA, April 12-14, 2011*, pages 211–220, 2011.

[106] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems *Specification**. Springer-Verlag, 1991.

[107] S. P. Miller. Will this be formal? *Theorem Proving in Higher Order Logics*, pages 6–11, 2008.

[108] S. Morimoto, S. Shigematsu, Y. Goto, and J. Cheng. Formal verification of security specifications with common criteria. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, 2007.

[109] S. Motre and C. Teri. Using b method to formalize the java card runtime security policy for a common criteria evaluation, 2000.

[110] NXP Semiconductors. *NXP Secure Smart Card Controller P60x144/080PVA, Security Target Lite*, May 2012.

[111] NXP Semiconductors. *NXP Secure Smart Card Controller P60D080/052/040yVC(Z/A)/yVG, Security Target Lite*, August 2014.

[112] Object Management Group. *Unified modeling language specification vers. 2.3*, 2010.

[113] Oracle. *Java Card 3 Platform, Application Programming Interface, Classic Edition, Version 3.0.4*, September 2011.

[114] Oracle. *Java Card 3 Platform, Runtime Environment Specification, Classic Edition, Version 3.0.4*, September 2011.

[115] Oracle. *Java Card 3 Platform, Virtual Machine Specification, Classic Edition, Version 3.0.4*, September 2011.

[116] D. Peled. Verification for robust specification. In *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, pages 231–241, 1997.

[117] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-based test case generation for smart cards. *Electr. Notes Theor. Comput. Sci.*, 80:170–184, 2003.

[118] N. Piterman and A. Pnueli. Faster solutions of Rabin and Streett games. In *Logic in Computer Science*, pages 275–284, 2006.

[119] N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380. Springer, 2006. LNCS 3855.

[120] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[121] N. Pontisso and D. Chemouil. TOPCASED Combining formal methods with model-driven engineering. In *Proc. Int. Conf. Automated Software Engineering*, pages 359–360, 2006.

[122] *Java Card Protection Profile - Open Configuration Version 3.0*, May 2012.

[123] R. Richards, D. Greve, and M. Wilding. The common criteria, formal methods and acl2. In *In Fifth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2004)*, 2004.

[124] R. Richards, D. Greve, and M. Wilding. A summary of intrinsic partitioning verification. In *In Fifth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2004)*, 2004.

[125] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[126] M. Rungger and P. Tabuada. Discounting the past in robust finite-state systems. In *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*, pages 842–847, 2014.

[127] R. Samanta, J. Deshmukh, and S. Chaudhuri. Robustness analysis of string transducers. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 427–441, 2013.

[128] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electrical Notes in Theoretical Computer Science*, 55(3):357–369, 2001.

[129] R. G. Schroeder. Programming solutions to ratio games. *Operations Research*, 18(2):300–305, 1970.

[130] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logic. *J. ACM*, 3(32):733–749, 1985.

[131] P. Tabuada, A. Balkan, S. Caliskan, Y. Shoukry, and R. Majumdar. Input-output robustness for discrete systems. In *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 217–226, 2012.

[132] P. Tabuada, S. Caliskan, M. Rungger, and R. Majumdar. Towards robustness for cyber-physical systems. *IEEE Trans. Automat. Contr.*, 59(12):3151–3163, 2014.

[133] P. Tabuada and D. Neider. Robust linear temporal logic. *arXiv preprint arXiv:1510.08970*, 2015.

[134] U. Topcu, N. Ozay, J. Liu, and R. Murray. On synthesizing robust discrete controllers under modeling uncertainty. In *Hybrid Systems: Computation and Control (part of CPS Week 2012), HSCC'12, Beijing, China, April 17-19, 2012*, pages 85–94, 2012.

[135] D. v. Oheimb, V. Lotz, and G. Walter. Analyzing SLE 88 memory management security using interacting state machines. *Int. J. Inf. Sec.*, 4(3):155–171, 2005.

[136] D. v. Oheimb, G. Walter, and V. Lotz. A Formal Security Model of the Infineon SLE 88 Smart Card Memory Managment. In *ESORICS*, pages 217–234, 2003.

[137] M. W. Whalen, D. D. Cofer, S. P. Miller, B. H. Krogh, and W. Storm. Integration of formal analysis into a model-based software development process. In *Formal Methods for Industrial Critical Systems, 12th International Workshop, FMICS 2007, Berlin, Germany, July 1-2, 2007, Revised Selected Papers*, pages 68–84, 2007.

[138] K. Wong, R. Ehlers, and H. Kress-Gazit. Correct high-level robot behavior in environments with unexpected events. In *Robotics: Science and Systems X, University of California, Berkeley, USA, July 12-16, 2014*, 2014.

[139] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.

[140] U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158:343–359, 1996.

Author Index

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……..
date (signature)