1 Introduction

Humans occasionally make mistakes, even programmers do.
Gerard J. Holzmann

The steadily growing complexity of computerized systems makes the latter highly susceptible to potentially critical design errors, ranging from simple programming bugs to scenarios and use cases overlooked by the engineers and therefore not properly handled by the system. The increasing use of computers in safety-critical applications like cars, aircrafts, power plants, or medical devices is therefore becoming a major concern. The risk entailed by delegating more and more safety-critical functions to computers is only acceptable if computing science manages to come up with new methods and tools which help to master the complexity of such systems and thus ensure their safety.

1.1 Formal Verification Using Satisfiability Checking

Simulation and testing, currently still being the most commonly practiced methods for design validation, allow to explore a relatively small number of executions of a system only and may thus fail to detect scenarios that expose a fatal bug. Except for trivial systems, simulation and testing are therefore insufficient to provide evidence for the correctness of a design.

An alternative approach for checking whether a design conforms to a desired property is formal verification. As opposed to simulation and testing, formal verification performs an exhaustive exploration of all possible behaviours of a system and is therefore guaranteed to detect existing errors or prove their absence. During the last ten years, formal verification of finite-state systems, such as digital circuits or communication protocols, has evolved from an academic subject to an approach accepted by the industry, with a number of commercial tools now being available and used by major companies.

The recent progress in SAT solving, i.e. the problem of deciding whether a given propositional formula has a solution, has motivated intensive research on how to exploit the strength of modern SAT solvers for formal verification. The result of this

C. Herde, *Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure* , DOI 10.1007/978-3-8348-9949-1_1,
© Vieweg+Teubner Verlag | Springer Fachmedien Wiesbaden GmbH 2011

effort are various verification algorithms, having in common that the verification task is reduced to the problem of checking the satisfiability of a propositional formula or a series thereof.

Arguably the most successful method in SAT-based verification of discrete systems is bounded model checking (BMC), as suggested by Groote et al. in [67] and by Biere et al. in [20]. In its most simple form, BMC is used to decide whether an unsafe system state is reachable via executions whose length is bounded by some given integer k. More precisely, BMC looks for a run of a finite transition system which starts in an initial state of the system, obeys the system's transition relation for k steps and ends in a state violating the safety property to be checked. The idea of BMC is to decide the existence of such a run, also referred to as error trace, by checking the satisfiability of the propositional formula

$$\phi_k = \bigwedge_{i=0}^{k-1} T(\vec{y}^{\,i}, \vec{y}^{\,i+1}) \;\wedge\; I(\vec{y}^{\,0}) \;\wedge\; R(\vec{y}^{\,k})$$

where $\vec{y}^{\,i}$ is the system's state-vector at step i of the run, $T(\vec{y}^{\,i}, \vec{y}^{\,i+1})$ encodes the transition relation, describing all admissible state transitions between steps i and $i+1$, and $I(\vec{y}^{\,0})$ and $R(\vec{y}^{\,k})$ are formulae characterizing the set of initial states and the set of unsafe states, respectively. By construction, ϕ_k is satisfiable if and only if an error trace of length k exists. The latter is checked by translating ϕ_k into conjunctive normal form (CNF) and solving it with a SAT solver. The solver not only outputs a yes/no result, but in case of satisfiability also delivers a satisfying assignment which directly corresponds to an error trace, i.e. a concrete scenario illustrating how the system can enter an unsafe state, thus providing a valuable debugging aid for the designer. BMC is usually carried out incrementally by constructing and solving ϕ_k for increasing values of k until either an error trace is found or some user specified limit on k, the solver's runtime, or its memory consumption is reached.

For every finite-state design there is a number k_c such that non-existence of an error trace of length less or equal than k_c implies that there is also no error trace of length greater than k_c. In theory, this indicates a way to make BMC complete, since ϕ_k has to be checked for $0 \leq k \leq k_c$ only. For this reason, k_c is called completeness threshold. In practice, however, k_c is usually too large to be reached by BMC and, moreover, cannot be computed efficiently [35]. BMC is therefore primarily seen as as method for falsification, i.e. for detecting bugs, rather than for proving their absence.

To cure the incompleteness of BMC, several techniques for *unbounded* model checking based on satisfiability checking have been proposed. Bjesse and Claessen

[24] and Sheeran et al. [109] suggest checking of safety properties using (temporal) induction schemes, where base case and induction step are encoded as Boolean formulae whose validity is checked with a SAT solver. Another approach, proposed by McMillan [89], uses Craig interpolants, which are derived from unsatisfiable BMC instances, to iteratively compute a propositional formula characterizing an overapproximation of the reachable state set of the system to be verified. To prove the safety of the system, it suffices to check whether the conjunction of this formula with a formula describing the set of unsafe states is unsatisfiable.

Albeit being limited to checking behaviours of bounded depth, BMC has proven to be a valuable tool for system validation. Its attractiveness is, amongst others, due to the fact that it has been found to scale better with the problem size than verification methods which use binary decision diagrams[1] to characterize the reachable state set [25], like classical symbolic model checking [31]. Enabled by the impressive performance gains of propositional SAT checkers in recent years, BMC can now be successfully applied even to very large finite-state designs [39].

1.2 Bounded Model Checking of Hybrid Systems

Often, safety-critical systems are not purely discrete, but their dynamics is characterized by an interaction of discrete and continuous behaviour. Such hybrid behaviour is found e.g. in the field of embedded systems where digital controllers are coupled with analog physical plants via sensors and actuators (see figure 1.1). Many properties of embedded controllers cannot be fully understood without taking into account the physical environment they are supposed to control. An analysis of such properties by means of formal verification requires a model of the combined hybrid discrete-continuous system.

As an example for a hybrid system consider a process control system of a chemical production process[2]. The state of the chemical plant is given by continuous variables, describing e.g. the fill level of tanks or temperature and pressure in various parts of the plant. Its evolution over time can be modeled by differential or difference equations. Each phase of the production process (e.g. evaporation of a fluid, draining a tank) corresponds to a discrete mode in the controller. Depending on the current mode, the controller opens or closes valves and activates or deactivates heaters and

[1]Binary decision diagram (BDDs) are graph-based data structures for efficient representation and manipulation of Boolean functions. For more information on BDDs, we refer the reader to [29].

[2]For concrete models of this kind see [82] and [54].

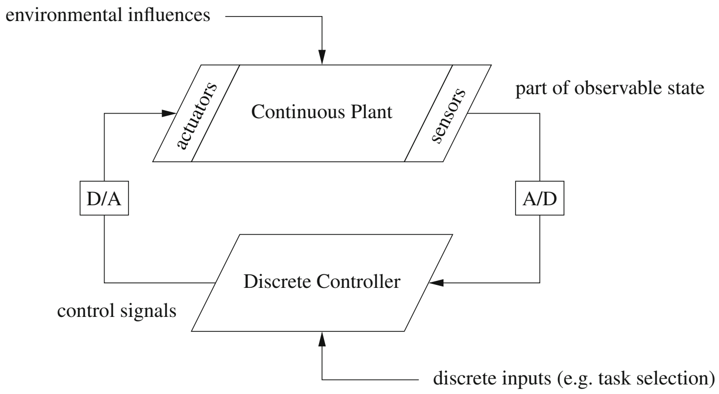

Figure 1.1: Standard set-up for a computer-controlled system.

pumps to control the respective phase of the process. Mode changes of the controller are triggered by progress of time or by state variables crossing specific thresholds (temperature of the fluid reaches boiling point, fluid level in the tank falls below a certain value). The continuous evolution of the chemical process is thus interleaved with discrete mode switches in the controller, yielding the typical discrete-continuous dynamics of a hybrid dynamical system.

However, hybrid dynamics not only arises from the use of digital control laws in a continuous physical environment. Often, a physical process itself has a natural hybrid representation. A four-stroke combustion engine, for example, is naturally modeled as a hybrid system[3]: While the four-stroke cycle of each piston can be described as a finite-state machine with four control modes, the individual phases intake, compression, expansion, and exhaust can each be modeled as a continuous process. Discrete and continuous dynamics are tightly coupled: The torque produced by each piston depends on the mode it is in. Conversely, mode transitions are triggered by the continuous dynamics of the power-train which, in turn, is driven by the engine torque.

A mathematical framework for modeling hybrid systems are hybrid automata. A hybrid automaton consists of a finite state machine where each control mode is equipped with continuous dynamics, usually specified by a set of differential equations, and a mode invariant, describing a region of the continuous state space that must not be left while staying in that mode. During a so-called flow, the continu-

[3]A detailed hybrid engine model like the one sketched here is described by Balluchi et al. in [8].

ous state changes according to the differential equations associated with the current control mode, while the discrete state (i.e. the control mode) remains unchanged. A discrete transition between control modes (also called jump) requires satisfaction of a certain guard condition and may also re-initialize continuous state variables. Guard conditions and mode invariants are usually expressed in form of arithmetic constraints over continuous state variables or inputs of the system.

Because of their relevance for industrial applications, hybrid systems have attracted a lot of interest in recent years. While tools for building hybrid models and for simulating their dynamics are commercially available (e.g. SIMULINK with the STATEFLOW extension[4]), theories and tool support for verifying hybrid systems are not yet mature. Current verification tools fall short with respect to both the dimensionality of the continuous state spaces and the size of the discrete state spaces they can handle. According to Asarin et al. [5] the *'analysis of systems with more than few continuous variables is considered very hard'*.

A promising approach to reduce the gap we observe between hybrid systems that can be specified and those that can be verified is to lift bounded model checking and related SAT-based methods for unbounded model checking to the hybrid domain. Though originally formulated for discrete transition systems only, the basic idea of BMC to reduce the search for an error path to a satisfiability problem of a formula also applies to hybrid discrete-continuous systems. However, the BMC formulae arising from such systems are no longer purely propositional. Encoding not only the discrete switching behaviour of the hybrid system, but also solution functions of differential equations, mode invariants and guard conditions, they usually comprise complex Boolean combinations of arithmetic constraints over real-valued variables, thus entailing the need for new decision procedures to solve them. The development of such procedures is the subject of this thesis.

1.3 Solvers for Boolean Combinations of Numerical Constraints

We address the problem of solving mixed propositional-numerical formulae like those arising as verification conditions from BMC of hybrid systems. We are thus aiming at formulae with the following characteristics:

The formulae are quantifier free and built from numerical constraints and propositional variables using arbitrary Boolean connectives. Numerical constraints are linear and nonlinear equalities and inequalites (strict or non-strict) involving variables

[4]SIMULINK and STATEFLOW are registered trademarks of The MathWorks, Inc.

of type real, integer and Boolean (the latter being interpreted as 0-1 variables when occuring inside arithmetic expressions). Besides polynomials, numerical constraints may involve transcendental functions, such as sine, cosine and the exponential function, which arise naturally e.g. in connection with solutions of differential equations. The formulae we address typically contain thousands of variables and thousands of constraints. Furthermore, they have a unique repetitive (we also say *symmetric*) structure since they comprise a k-fold unwinding of a transition relation.

Given such a formula, our algorithmic goals are to check the satisfiability of the formula and, in case of satisfiability, determine a sample solution, i.e. a variable assignment under which the formula evaluates to true. In the context of BMC, the sample solution will provide us with an error trace, i.e. a scenario refuting the property under verification.

1.3.1 Constraint Solving in a Nutshell

Solving formulae is a basic activity in mathematics and computing, yet one which is inherently difficult for many formula classes of practical relevance. From an algorithmic point of view, the problem of solving Boolean formulae and the problem of solving arithmetic constraint systems have been treated separately for a long time, resulting in specialized algorithms for the respective tasks. Solvers which combine algorithms for Boolean and numerical constraints first appeared in the 1990s.

Boolean Formulae

The problem of deciding the satisfiability of a Boolean formula, also referred to as SAT problem, was shown to be NP-complete by Stephen Cook in 1971 [38]. Most modern SAT solvers implement variants of the Davis-Putnam-Logemann-Loveland (DPLL) procedure which was devised already in the early 1960s [43, 42]. DPLL requires the input formula to be in conjunctive normal form (CNF), i.e. a conjunction of clauses, where a clause is a disjunction of literals, the latter being a Boolean variable or its negation. Hence, the problem has to be translated into this format first. The DPLL procedure aims at incrementally extending a partial variable assignment towards a satisfying assignment by alternately performing logical deduction, which involves an exhaustive application of the so-called unit propagation rule, and branching, i.e. tentatively assigning a truth-value to a previously unassigned variable, thus providing new input to the subsequent deduction phase. In case of a conflict, i.e. if the current partial assignment violates the input formula, DPLL backtracks to an earlier choice point and tries another branch. This is repeated until either a solution is found or backtracking

Satisfiability	over $\mathbb{R}$	over $\mathbb{Z}$	over $\mathbb{N}$	over $\mathbb{B}$
Linear equations	polynomial	polynomial	NP-compl.	NP-compl.
Linear inequations	polynomial	NP-compl.	NP-compl.	NP-compl.
Polynomial equations	decidable	undecid.	undecid.	NP-compl.
Polynomial inequations	decidable	undecid.	undecid.	NP-compl.
Transcendental equations	undecid.	undecid.	undecid.	NP-compl.
Transcendental inequations	undecid.	undecid.	undecid.	NP-compl.

Table 1.1: Solvability of numerical constraints.

runs out of branches in which case the formula is unsatisfiable. State-of-the-art SAT solvers enhance this basic procedure with numerous algorithmic improvements, such as lazy clause evaluation for speeding up the deduction process and non-chronological backtracking which involves an analysis of the reason for a conflict in order to back up directly to the earliest choice which caused the conflict. The latter is typically used in connection with conflict-driven learning, a technique which adds so-called conflict clauses to the input formula in order to avoid repeated failures due to the same reason. Modern SAT solvers which incorporate these enhancements are capable of solving CNFs with hundreds of thousands of Boolean variables and clauses. See [126] for a more detailed report on techniques employed in modern SAT solvers.

Numerical Constraints

Decision procedures for numerical constraints usually solve systems of *simultaneous* constraints, i.e. sets of constraints which are implicitly assumed to be conjoined. An extensive overview of solving algorithms for various classes of numerical constraints is given in [26]. In terms of computational complexity, the problem of checking the satisfiability of a set of simultaneous numerical constraints ranges from polynomial to undecidable, depending on the domains of the variables and the operations involved. Table 1.1 summarizes the state of affairs.

Systems of linear equations over the reals can be solved in polynomial time using Gaussian elimination. If also linear inequalities over the reals are involved, procedures for linear programming (LP) like the simplex algorithm can be used. The linear programming problem was shown to be solvable in polynomial time by Leonid Khachiyan in 1979 [80]. Although the simplex algorithm is known to have a worst-case exponential behaviour, this complexity does very rarely manifest itself in practice. Sophisticated implementations are capable of solving problems with tens of thousands constraints and millions of variables. If some or all variables of a linear system are required to take integer values only, the complexity of deciding its satisfiability jumps from polynomial to NP-complete. Extensions of LP to mixed integer linear programming (MILP), where not all variables are required to be integer, and to pure integer linear programming (ILP) are typically limited to problems involving a few hundred or thousand integer variables. We note that CNF-SAT is a special case of solving linear systems over the integers, since each CNF clause can be translated into a linear inequality over 0-1 variables. The clause $x \vee \overline{y}$, for example, can be written as $x + (1 - y) \geq 1$.

The solvability of systems of polynomials over the reals follows from the decidability of elementary geometry which was proved by Tarski in 1948 [114]. The first decision procedure with elementary complexity, called cylindrical algebraic decomposition(CAD), was devised by Collins in 1975 [37]. The idea underlying CAD is to decompose the solution space into a finite number of so-called cells such that every polynomial has a constant sign in each cell. The worst-case number of cells, however, is doubly exponential in the number of variables occuring in the input formula such that CAD can only be applied to very small problems. Due to the undecidability of Hilbert's tenth problem [88], there is no similar solving algorithm for systems of polynomials over the integers. For the very same reason, the satisfiability of transcendental constraints over the reals is undecidable, because the periodicity of trigonometric functions can be used to filter out a model of the integers from the reals.

A more practical approach to tackle polynomial and transcendental constraints is interval constraint solving (ICS) [17]. ICS calculates a finite set of interval boxes whose union is an overapproximation of the solution set. To this end, ICS recursively refines an initial box-covering of the search space by applying interval-arithmetic deduction rules to boxes, thereby pruning parts of the interval boxes which contain non-solutions only, and by splitting boxes into sub-boxes if otherwise no further pruning can be achieved. The minimal width of a box which is selected for splitting controls the tightness of the overapproximation.

Boolean Combinations of Numerical Constraints

Checking the satisfiability of a propositional formula involving numerical constraints is an instance of the so-called Satisfiability Modulo Theories (SMT) problem, i.e. the problem of deciding the satisfiability of a quantifier-free formula which is a Boolean combination of propositions and atomic formulae over some background theory T.

Currently, the most effective approach to SMT is to combine a propositional SAT solver with a decision procedure for conjunctions of atoms from the respective background theory as follows. The SAT engine enumerates satisfying assignments of a Boolean abstraction of the input formula which is obtained by replacing each theory constraint with a new propositional variable representing the truth value of the respective constraint. A satisfying assignment of the Boolean abstraction thus defines a set of T-atoms that have to be satisfied simultaneously in order to satisfy the original formula. This is checked by the T-solver. If the set of T-constraints turns out to be inconsistent, an explanation for the conflict (usually a subset of T-constraints which causes the inconsistency) is used to refine the Boolean abstraction. The refinement process is iterated until either the Boolean abstraction (and, hence, the input formula) turns out to be unsatisfiable or a T-consistent satisfying Boolean assignment is found, in which case the input formula is satisfiable as well. The use of a DPLL-based SAT solver enables an optimization of the above scheme which exploits the incremental construction of an assignment in the DPLL procedure. Instead of checking only complete propositional assignments against the background theory, the T-consistency of a truth assignment is already verified for partial assignments, e.g. after each deduction phase of the DPLL solver. This allows an early pruning of the Boolean search tree. The resultant solver architecture is called DPLL(T).

The solver integration sketched above is often referred to as the *lazy* approach to SMT since inconsistencies between T-constraints are detected and learned on the fly during DPLL proof search. As opposed to this, the *eager* approach uses the T-solver prior to the SAT search in order to perform a satisfiability-preserving translation of the input formula into a purely propositional formula which is then solved by a SAT solver. The eager approach, however, often suffers from a massive blow-up of the formula size during the translation. This is in particular true for arithmetic theories, which renders the eager approach impractical for input formulae containing more than a small number of theory constraints. In-depth surveys of current SMT technology are given in [12] and, with a focus on the lazy approach, in [108].

1.3.2 Contributions of the Thesis

The research we report in this thesis contributes to the state-of-the-art in solving mixed propositional-numerical formulae in a number of aspects. We present our contributions by addressing formula classes of increasing expressiveness, starting out with purely propositional formulae, then advancing to Boolean combinations of *linear* equalities and inequalites, and finally dealing with Boolean combinations of *nonlinear* (and transcendental) constraints. Although being dedicated to the task of verification of hybrid systems, most of the techniques we describe are general purpose and have applications in many other domains, like operations research, planning, software verification, and scheduling, for example.

Contribution 1: Acceleration of DPLL for Pseudo-Boolean Constraints

A key component in the solving algorithms we present in this thesis is the DPLL procedure for checking CNF satisfiability. The algorithms we propose extend and generalize the DPLL procedure. Hence, any improvement made to DPLL directly carries over to the performance of the solvers for mixed propositional-numerical formulae. Our first contribution therefore addresses, as a start, propositional satisfiability and investigates the problem of generalizing acceleration techniques as found in recent satisfiability engines for conjunctive normal forms (CNFs) to systems of linear constraints over the Booleans, also called pseudo-Boolean constraints. The motivation for this research is that compared to clauses of a CNF, pseudo-Boolean constraints allow a significantly more compact description of many discrete problems. Their modeling power has been widely used in the fields of logic synthesis, operations research, and formal verification, see [3] and [30] for example. We demonstrate that acceleration techniques like lazy clause evaluation, non-chronological backtracking and learning techniques generalize smoothly to Davis-Putnam-like procedures for the very concise propositional logic of linear constraint systems over the Booleans. Despite the more expressive input language, the performance of our prototype implementation comes surprisingly close to that of CNF-SAT engines like CHAFF [91]. Experiments with bounded model-construction problems show that the overhead in the satisfiability engine that can be attributed to the richer input language is often amortized by the conciseness gained in the propositional encoding of the BMC problem.

Contribution 2: BMC-Specific Optimizations in a DPLL(LA) Solver

As a second step, we address the problem of solving Boolean combinations of constraints from the theory of linear arithmetic over the reals (LA). This subclass of formulae is especially appealing because it is decidable, i.e. allows for a complete solving algorithm, and it is powerful enough to model hybrid automata with linear dynamics. The latter are of particular practical relevance since nonlinear physical systems can often be approximated with sufficient accuracy by piecewise linear approximations. We aim at providing a solver which is tailored for BMC of linear hybrid automata. To this end, we propose a DPLL(T) architecture where the T-solver is instantiated with a linear programming (LP) routine which is used to decide the feasibility of sets of linear constraints. We investigate how to take advantage of the specific properties of linear programming to make the coupling of DPLL and LP as efficient as possible. In particular, we study various methods for computing small (ideally minimal) infeasible subsystems of a conflicting linear constraint system which serve as explanations for arithmetic conflicts. Explanations are learned by the SAT solver in form of conflict clauses which are used to resolve the conflict and in order to prevent the solver from running into further arithmetic conflicts due to the same reason. We examine empirically the impact of these methods on the runtime of the DPLL(LA) engine. Furthermore, and most importantly, we exploit the unique characteristics of BMC formulae (in particular their symmetry) and the incremental nature of BMC for a variety of optimizations in the solver. We demonstrate that these optimization, which were originally proposed by Ofer Strichman for BMC of purely discrete systems [113], pay off even better in the hybrid domain. The reason is that an inference step involving arithmetic constraints is computationally much more expensive than one involving propositional logic only. To the best of our knowledge, our solver was the first to make use of BMC-specific optimizations in order to accelerate solving of formulae arising from verification of hybrid systems.

Contribution 3: Integration of DPLL and Interval Constraint Solving

Finally, we address the problem of solving large Boolean combinations of nonlinear arithmetic constraints involving transcendental functions. For this purpose, we propose a tight integration of a DPLL-based SAT solver and interval-arithmetic constraint solving. The undecidability of the arithmetic base theory, however, precludes the use of a DPLL(T) approach since DPLL(T) heavily relies on the availability of a *complete* theory solver. Instead, we exploit the algorithmic similarities between DPLL-based propositional SAT solving and interval constraint solving for a much tighter integration, where the DPLL solver directly controls arithmetic constraint propagation rather than delegating arithmetic inferences to a subordinated theory solver. The resulting solving algorithm, which we refer to as iSAT, turns out to be an elegant generalization of the DPLL routine. It inherits the branch-and-deduce framework and the unit propagation rule for logical deductions from DPLL and adds deduction rules for arithmetic operators which are adopted from interval constraint solving. Through this tight integration, all the algorithmic enhancements that were instrumental to the enormous performance gains recently achieved in propositional SAT solving carry over smoothly to the rich domain of nonlinear arithmetic constraints. We demonstrate that our approach is able to handle large constraint systems with complex Boolean structure, involving Boolean combinations of multiple thousand arithmetic constraints over some thousands of variables. Having the structure of a DPLL routine, the iSAT algorithm can be extended with all algorithmic contributions discussed in the previous sections. It can be tuned for BMC using Strichman's optimizations, it can be equipped with a special deduction rule for pseudo-Boolean constraints, and a lazy integration of linear programming, yielding an iSAT (LA) solver, will strengthen iSAT's deductive power for linear constraints. We consider the iSAT algorithm to be the main contribution of this thesis.

1.3.3 Structure

Subsequent to this introduction, chapter 2 sets out to give some insight into the origin of the formulae we are concerned with. After recalling some basic definitions on hybrid automata, we demonstrate by means of a running example various ways to encode the next-state relation of a hybrid automaton as a formula involving Boolean and arithmetic constraints.

The following three chapters present the contributions explicated above. Chapter 3 deals with extending DPLL for pseudo-Boolean constraints, chapter 4 investigates the coupling of DPLL with linear programming and tuning the resulting solver for BMC, and chapter 5 describes the iSAT algorithm for solving Boolean combinations of nonlinear constraints. Each chapter first defines the formula class to be tackled and thereafter reviews the preliminaries which are required to prepare the reader for the subsequent exposition of the algorithmic contributions. A final section reports on the benchmarks conducted to evaluate specific aspects and the overall performance of the methods proposed in the respective chapter. In particular, a case study of a controller for separation of trains being operated under a moving block principle is presented in chapter 5 in order to demonstrate the applicability of the iSAT approach to designs from the envisaged application domain.

Chapter 6 provides a summary of the results of the thesis, followed by suggestions for future research.

1.3.4 Sources

This thesis draws from the following conference and journal publications of the author. Chapter 3 is based on a conference paper published at LPAR'03 [60]. The article which forms the basis of chapter 4 received the EASST best paper award at FMICS'04 and was published in *Electronic Notes of Theoretical Computer Science* in 2005 [61]. A blend of material from [60] and [61] appeared 2007 in *Formal Methods in Systems Design* [62]. Chapter 5 is a revised and extended version of an article published in 2007 in the *Journal on Satisfiability, Boolean Modeling and Computation* [63]. The train case study presented in chapter 5 was published at ICONS'08 [74].

The contributions and achievements of this thesis, though original work by myself, have obviously been influenced and shaped in many discussions with the co-authors of the above papers and many other researchers. In particular, I benefited a lot from the exchange and tight collaboration with friends and colleagues from the *University of Freiburg*, *Saarland University*, the *Academy of Sciences of the Czech Republic*, and the *University of Oldenburg* within the Transregional Collaborative Research Center 'Automatic Verification and Analysis of Complex Systems' (AVACS), funded by the DFG.