SAT-Solving in Practice, with a Tutorial Example from Supervisory Control

Discrete Event Dynamic Systems manuscript No.
(will be inserted by the editor)

SAT-solving in practice, with a tutorial example from supervisory control

Koen Claessen · Niklas Een · Mary Sheeran · Niklas Sörensson · Alexey Voronov · Knut Åkesson

A post-print for: Discrete Event Dynamic Systems (2009), Volume 19, Issue 4, pp 495-524 (see doi 10.1007/s10626-009-0081-8).

Abstract Satisfiability solving, the problem of deciding whether the variables of a propositional formula can be assigned in such a way that the formula evaluates to true, is one of the classic problems in computer science. It is of theoretical interest because it is the canonical NP-complete problem. It is of practical interest because modern SAT-solvers can be used to solve many important and practical problems. In this tutorial paper, we show briefly how such SAT-solvers are implemented, and point to some typical applications of them. Our aim is to provide sufficient information (much of it through the reference list) to kick-start researchers from new fields wishing to apply SAT-solvers to their problems. Supervisory control theory originated within the control community and is a framework for reasoning about a plant to be controlled and a specification that the closed-loop system must fulfil. This paper aims to bridge the gap between the computer science community and the control community by illustrating how SAT-based techniques can be used to solve some supervisory control related problems.

Keywords: formal verification, Boolean satisfiability problem, model checking, supervisory control.

K. Claessen, M. Sheeran and N. Sörensson
Department of Computer Science and Engineering,
Chalmers University of Technology, SE-41296 Göteborg, Sweden
E-mail: {koen,ms,nik}@chalmers.se

A. Voronov and K. Åkesson
Department of Signals and Systems,
Chalmers University of Technology, SE-41296 Göteborg, Sweden
E-mail: {voronov,knut}@chalmers.se

N. Een
Cadence Research Labs,
2150 Shattuck Avenue, 10th Floor
Berkeley, CA 94704, USA
E-mail: niklas@cadence.com

1 Introduction

Given a propositional formula, the *Boolean Satisfiability* or *SAT* Problem is to determine whether there exists a variable assignment such that the formula evaluates to true. This is the classic NP-complete problem (Cook, 1971), and has therefore attracted much attention from researchers. In this tutorial paper, we explain SAT and how it can be implemented, and give pointers to how it is used in practice, including many references. As well as introducing some well established applications of SAT (for example to hardware verification), the paper extends an earlier version (Claessen et al., 2008) with a case study on the formalisation of Supervisory Control Theory in SAT. Our hope is to attract researchers from new fields, and particularly those related to the analysis of discrete event systems, to explore applications of SAT.

1.1 Why SAT is interesting from a practical point of view

The fact that SAT-solving is hard on average in no way precludes its use in solving the particular SAT instances that arise in real problems. Recent progress in practical applications of SAT has built upon two bases: improved SAT-solving engines and innovative ways to encode real problems in ways that can exploit those engines. Recent SAT-solvers have been developed in a scientific community that has placed great store in practical applicability, and the development of the solvers has in turn spurred work on new ways to exploit such solvers. The resulting positive spiral has led, for instance, to the development of commercial hardware verification tools in which SAT-solvers, and algorithms that use SAT-solvers, are a vital component.

1.2 State of the art until 1999

Around 1999, symbolic manipulation of boolean functions had long been important in circuit synthesis and verification. Work in this area concentrated on the use of Binary Decision Diagrams (BDDs) (Bryant, 1986). A glance at the conference proceedings of the first international conference on Formal Methods in Computer Aided Design of Electronic Circuits (1996) (Srivas and Camilleri, 1996) makes this very clear.

Before this time, SAT was largely a theoretical subject, with a few notable exceptions. One of the classic application areas for SAT has been planning (Kautz and Selman, 1992). In electronic design automation, some typical applications that began to appear in the 1990s were timing analysis (Silva et al., 1998), test pattern generation (Larrabee, 1992; Marques-Silva and Sakallah, 1997) and FPGA routing (Nam et al., 1999). Motivated by the problem of generating test vectors for combinational and sequential circuits, Kunz and Pradhan introduced the notion of *recursive learning* and demonstrated that it had application not only in test generation, but also in optimization and verification (Kunz and Pradhan, 1994). Indeed, recursive learning could, in the early nineties, be seen as a new approach to the boolean satisifiability problem. Stlmarck's patented method of SAT-solving (Sheeran and Stålmarck, 1998) was used during the 1990s in the formal verification of railway signalling systems (Borälv and Stålmarck, 1998; Groote et al., 1995; Säflund, 1994; Stålmarck, 1990); indeed the method is still used commercially for this purpose, and can in fact be seen as an early version of Bounded Model Checking (introduced in Sect. 3). The formulas that result from such verification are truly gigantic,

but Stlmarck's method copes well with the large but easy SAT instances that result. This work on railway signalling verification, perhaps because of its scalability, was one of the first really successful projects in practical, industrial, formal methods.

Also in an industrial setting, Siemens Corporate Research, already in the 1980s, started a major initiative to explore the potential of formal methods for the company's own products and systems. For circuit design verification, a particularly successful solution called Circuit Verification Environment (CVE) was developed. In the mid 1990s the basic methodology and proving machinery of CVE was radically changed from symbolic (BDD-based) model checking to a SAT-based approach. The property language Interval Temporal Logic (ITL) was developed, which expresses system behaviour over bounded time intervals. Proving such bounded properties can be mapped to a SAT instance. This new paradigm was quickly adopted in Siemens design flows and became standard practice already in 1996/1997, a few years before academic work on bounded model checking was published. The success of this approach triggered intensive efforts to improve SAT-solving procedures in Siemens, and later at Infineon and OneSpin (the spin-off company that now develops and markets the technology). In parallel, Kunz was developing similar ideas about checking bounded intervals, working with Mentor (a major tool-vendor in Electronic Design Automation). At this time, industry was the driving force in innovation in applied formal methods, and unfortunately the work was not published; but the impact of academic research was soon to increase. Developed in academia, the GRASP (Marques-Silva and Sakallah, 1996) and SATO (Zhang, 1996) solvers were early examples of solvers intended for use on large-scale problems, and they influenced later developments.

1.3 The SAT revolution

In 1999, the notion of *Bounded Model Checking (BMC)* was published, and immediately recognised to be of great practical interest (Biere et al., 1999a). It can be thought of as checking that a property holds not for all possible behaviours of a system but only for the first n steps (from the initial states), for a given fixed n. This apparently simple idea has proved extremely effective in practical hardware verification, and is now included in all formally-based commercial tools for hardware verification. It will be considered in more detail in section 3.

At around the same time, the high-performance SAT-solver Chaff became available, and was widely used by researchers in applications of SAT (Moskewicz et al., 2001). For two of the authors (Een and Sörensson), seeing a presentation about Chaff provided inspiration to build small well-structured, yet high-performance, solvers, culminating in miniSAT (Een and Sörensson, 2004). This solver and its associated description can act both as a tutorial and as a starting point for researchers wishing to modify it for their purposes. One of the driving forces in the development of efficient solvers has been the international SAT competition, which has led to the creation of benchmark sets, and also makes the competing solvers available to the research community (Sat). Section 2 presents the basics of SAT-solving.

Independently of the developments in BMC, researchers from Chalmers worked with Stålmarck, who proposed a form of induction for use in complete (rather than bounded) model checking (Bjesse and Claessen, 2000; Sheeran et al., 2000). The method, and also BMC, demands satisfiability checking of many related SAT-instances, which in turn led to an *incremental* version of miniSAT (Een and Sörensson, 2003). An incremental

SAT-solver supports an programming interface that allows the SAT-problem to be modified and queried dynamically, rather than solving one SAT-problem at a time. Section 4 presents the basics of *temporal induction*.

1.4 The Supervisory Control Approach

Within the control community the general approach is to build a model of the process to be controlled. The engineer then generates specifications, typically performance and robustness specifications, that the plant has to fulfill when controlled by a computer controller implementing some control algorithm. The control design task is to design the control algorithm such that the specifications are fulfilled given a model of the process to be controlled. In classic control the processes to be controlled are typically electrical, mechanical or chemical, modeled using differential equations. A similar approach can be used for discrete-event systems modeled using finite automata. The theoretical foundation for this is known as supervisory control theory (Cassandras and Lafortune, 2008; Ramadge and Wonham, 1987, 1989). The *supervisor* is a safety device that may prevent certain events from occurring in the plant, i.e. the process to be controlled. The plant it assumed to generate all events. Some events are controllable and may be disabled by the supervisor and some events are uncontrollable and may be spontaneously generated by the plant and thus cannot be disabled by a supervisor.

The supervisory control problem can be divided into a verification problem and a synthesis problem. In the verification problem, the task is to verify if a supervisor that is acting together with a plant fulfills given specifications. In the synthesis problem, the task is to generate a supervisor, given models of the plant and the specifications, in such a way that the closed-loop system fulfills the given specifications. Usually it is also required that the supervisor should do this by restricting the plant as little as possible. The supervisory approach can, for example, be used in manufacturing systems where both the resources available to produce a part and the products to be produced are changed frequently, so that the control functions need to be updated frequently. However, the supervisory control problems suffer from *state-space explosion* and different techniques have been used to handle problems of industrial size, like modular and compositional algorithms for dividing the problems into smaller sub-problems, and binary decision diagrams for efficiently representing large state-spaces. In Section 5, which is an extension of (Voronov and Åkesson, 2008), we show how to solve some supervisory problems using SAT-based techniques, thus illustrating some of the SAT-techniques introduced earlier in the paper.

2 The basics of a modern SAT-solver

2.1 Formal Definition of The SAT Problem

A propositional logic formula is said to be in CNF, *conjunctive normal form*, if it is a conjunction ("and") of disjunctions ("ors") of literals. A literal is either x, or its negation $\neg x$, for a boolean variable x. The disjunctions are called *clauses*. The satisfiability (SAT) problem is to find an assignment to the boolean variables, such that the CNF formula evaluates to true. An equivalent formulation is to say that *each clause* should have at least one literal that is true under the assignment. Such a clause is then said to be

satisfied. If there is no assignment satisfying all clauses, the CNF formula is said to be *unsatisfiable*.

Propositional formulas that are not in CNF can be transformed into CNF in a standard way (Marques-Silva, 2008; Tseitin, 1968), a process that is called *clausification*. In general, extra variables have to be introduced to keep this transformation linear in time and size. Clausification is still an active research area, see for example (Een et al., 2007).

2.2 Boolean Constraint Propagation

During the search for a satisfying assignment, the solver will maintain a *partial assignment*, with some variables assigned to either 0 or 1, and others still unassigned. For a given partial assignment, a clause may find that all its literals except one are false. When this happens, the only way to satisfy that clause is to *fix* (or assign) the variable of the last literal to the appropriate value that makes the literal true. This observation defines a process of deriving new variable assignments from the current partial assignment. It can be implemented very efficiently (Moskewicz et al., 2001), and when run to saturation (no more assignments can be derived) is referred to as *Boolean Constraint Propagation* (BCP), also called unit resolution.

Example 1 (Boolean Constraint Propagation) Consider the following three clauses:

$$\{\text{-a, b}\}\ ,\ \ \{\text{-a, c}\}\ ,\ \ \{\text{-b, -c, d}\}$$

If the partial assignment consists of one fixed variable "a=1", then from the first two clauses, BCP will derive "b=1" and "c=1"; which in turn will imply, through the last clause, "d=1".

2.3 Conflicts, Learning, and Backtracking

A simple and complete SAT algorithm can be achieved by a standard backtracking search: pick an unassigned variable, fix it to either 0 or 1 (this is called a *decision*) and recursively solve the resulting subproblem. If no solution was found, flip the variable to the other value and recurse again. After each branching, the partial assignment is investigated to see if there is an unsatisfied, or *conflicting*, clause (all literals are false). If so, there is no need to branch further (return NoSolution). If on the other hand all variables have been fixed without a conflict, a satisfying assignment has been found.

The procedure can be improved by running BCP after each fixed variable to get all the cheap implications. This procedure, backtracking + BCP, is commonly referred to as DPLL (after David, Putnam, Logemann and Loveland, the combined authors of two classic SAT papers (Davis and Putnam, 1960; Davis et al., 1962)) and until the inception of modern SAT solvers was the predominant approach to SAT.

Modern SAT, through a series of improvments to DPLL, has been refined to an algorithm that is sufficently different from the original to deserve its own name. We will refer to it as *Conflict Driven SAT Solving* (CDSS). It differs from DPLL in three important respects:

1. It is not a recursive procedure. Instead an explicit stack of assignments (referred to as the *trail*) is used for backtracking.
2. It derives and adds new clauses through a learning mechanism. This procedure takes place each time a conflicting clause is detected during the search. The added clauses are redundant in the sense that the resulting problem is logically equivalent, but they assist the BCP in fixing literals throughout the remainder of the search.
3. Backtracking is no longer restricted to return to the previous decision. The outcome of clause learning is actually two-fold: while producing a learned clause, it also analyses which of the decisions contributed to the conflict. If the k latest decisions were irrelevant for the conflict, the procedure will undo all those k decisions (and their BCP implications) rather than just the last.

Putting it all together in pseudo-code, the modern SAT algorithm is:

```
forever {
    bcp
    if no conflict {
        if no unassigned variable { return SAT }
        make decision
    } else {
        if no decisions were made { return UNSAT }
        analyze conflict
        undo assignments
        add learned clause
    }
}
```

For a more detailed description of this procedure, see references (Een and Sörensson, 2004; Moskewicz et al., 2001); a list of improvements can be found in (Een, 2007).

2.4 Making decisions

The key to making the above algorithm effective is to tie the variable decision heuristic to the clause learning. This is done by increasing the so called *activity* of all the variables present in any of the clauses contributing to a conflict. It will bias the search to stay in the region of the most recent conflicts while ignoring variables that were not involved in those conflicts. In effect, the heurstic forces the solver to exhaust all possible conflicts in a subregion, typically resulting in a set of short, learned clauses that captures, more concisely than the original clauses, the reason why that region of the search space is unsatisfiable. To further focus the search, all activites are *decayed*, in other words periodically multiplied with a number < 1, to give higher weight to more recent conflicts. This variable heuristic VSIDS (Variable State Independent Decaying Sum) has emperically been proven to successfully localize large industrial SAT problems and solve them by homing in on the relevant part (Shacham and Zarpas, 2003).

2.5 State of the art in SAT

Improving on the state of the art of SAT has turned out to be a really hard task, as indicated by the slow progress made this century in the development of core SAT

algorithms. Since the beginning of the SAT revolution, research effort has been mostly directed towards investigating new applications, possible extensions, and exploring different techniques for encoding particular problems. For instance, looking at the papers accepted to SAT'06 there is barely a single paper that can be considered to attempt to improve on the core algorithms of a DPLL type SAT solver.

Here is a list of noteworthy work that has improved upon core SAT technology since the appearance of Chaff in 2001 (Moskewicz et al., 2001).

Conflict clause minimization (2005) – an improvement in the conflict clause construction algorithm which generally makes conflict clauses stronger, and therefore the proof search more efficient (Een and Sörensson, 2005).

Variable-elimination-based preprocessing (2005) – most practical SAT problems can be greatly simplified before being fed to a SAT solver, reducing their size and complexity, and subsequently reducing solving time; this was first implemented in the tool SatELite (Een and Biere, 2005).

Improvements to decision heuristics (2002-2007) – a lot of work has gone into heuristics for how to choose the next variable to branch on in the proof search, and what value it should have first (Goldberg and Novikov, 2002; Pipatsrisawat and Darwiche, 2007), with visible effects on solving efficiency for industrial problems.

Improvements to restart heuristics (2007) – modern SAT solvers, in order to avoid getting stuck in one corner of the search space, perform a "restart" every once in a while in the middle of a search; some search parameters are reset and the search restarts at top-level. Special heuristics have been developed for when and how often to do this during search (Huang, 2007; Luby et al., 1993).

Data structure improvements for clauses (2000-2007) – new datastructures have been developed for representing clauses; e.g. improving the representation of clauses with two literals (Een and Sörensson, 2005), and improving memory access patterns in general (Biere, 2007a,b).

The next two sections describe two particular and very common applications of SAT-solvers in formal verification of properties of systems, namely Bounded Model Checking and Temporal Induction.

3 Bounded Model Checking

The most widespread use of SAT-solvers in industrial property-based system verification today is in *Bounded Model Checking*. Model checking (in general) is an automated technique for checking if a given implementation of a system satisfies a given property, specified in some logic. The answer can either be "yes", in which case the property holds, or "no", in which case the model checker produces a counter-example to the property at hand. A counter-example is a concrete *path* (i.e. a sequence of consecutive states that starts in the initial state) for which the desired property is false.

Up to the late 1990s, model checking for hardware systems was dominated by *symbolic* model checking methods based on Binary Decision Diagrams (BDDs) (Bryant, 1986), a data structure providing a canonical representation for boolean formulas. BDDs can be used for representing sets of reachable states of a system. A model checker computes the set of all reachable states by a repeated use of boolean variable quantification, an operation that is well supported by the BDD data structure. Though rather successful for certain classes of circuits, BDD-based model checkers suffer from a potential *BDD-blowup*, when the size of the BDD data structures becomes too large to

handle in memory. Many techniques have been developed for battling BDD-blowup in certain situations, but the actual problem in general remained.

At the end of the 1990s, several research groups were independently trying to alleviate the problem of BDD-blowup by replacing BDDs by other technologies. For example, the model checker FixIt (Abdulla et al., 2000; Een, 1999), developed by Abdulla, Bjesse and Een, replaced BDDs in the standard model checking algorithms by regular non-canonical propositional formulas. They developed their own cunning variable quantification algorithm, and used a SAT-solver to reason about the formulas. The result was a model checker that complemented the existing BDD-based model checkers. Unfortunately, the variable quantification turned out to be a memory bottle neck, often leading to excessive memory usage.

Bounded Model Checking (BMC) was first presented at the conference on Formal Methods in Computer Aided Design (FMCAD) in 1998. In an unprecedented move, the Chairs of FMCAD'98 permitted the inclusion of an extra talk about BMC. This was followed up by several publications in 1999, including the first paper outlining the idea (Biere et al., 1999a) and one describing its application at Motorola to the verification of a PowerPC processor (Biere et al., 1999b).

A BMC model checker is parametrized by a natural number, a *bound* n, and only tries to find counter-examples (paths) that consist of no more than n transitions. The answer a BMC model checker thus produces is "no", with a counter-example, or "could not find a counter-example of length n or smaller". The big gain of this approach is that the BMC model checker does not have to perform variable quantification, the precise thing that was the bottle neck for the model checker FixIt.

Let us look concretely at how a BMC model checker works. For simplicity, we assume that the property we are checking is a so-called *simple safety property*. This means that given a single state of the system, we can decide if this is a good state (the desired property holds) or a bad state (the property does not hold). The original BMC paper presents a symbolic, SAT-based algorithm that can deal with more general properties than these (Biere et al., 1999a).

To model the system under verification and the property in the BMC framework, the state of the system is represented by a finite vector of boolean variables s. The safety property is represented by a formula $P(s)$, which is true precisely for the good states, the states where the property holds. To model the system itself, we split it up into two parts: the initial states and the transitions between the states. The set of initial states is modelled as a formula $I(s)$, which is true if and only if s represents an initial state of the system. Finally, the transitions of the system are modelled by a formula $T(s, s')$ that is true if and only if the system can make a transition between the states represented by s and s'.

Now, in order to check whether or not there exist counter-examples of length n or smaller, we create a sequence of vectors of boolean variables $s_0, s_1, \ldots, s_n$, and build the following formula:

$$\begin{aligned} I(s_0) \wedge (T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{n-1}, s_n)) \\ \wedge (\neg P(s_0) \vee \neg P(s_1) \vee \cdots \vee \neg P(s_n)) \end{aligned}$$

The above formula expresses restrictions on the values of the variables in s_i; namely that s_0 should be an initial state, that there must be a transition from s_j to s_{j+1} for all $j < n$, and that at least one of the states visited must be a bad state. Any satisfying assignment to the above formula therefore represents a counter-example to the property.

A BMC model checker now simply invokes a SAT-solver to check whether or not that is the case.

The original BMC paper (Biere et al., 1999a) also discusses ways of finding out how large n should be in order to be sure that no counter-example of any length can be found. This so-called *diameter* turned out to be expensive to compute in practice.

BMC is very successful at finding bugs. The user can start with small values of n, for which the method is very cheap, and successively increase n when no counter-examples are found. In this way, very quick feedback is provided about the status of properties, without having to perform a full general model checking procedure. This has led to a paradigm shift in industrial applications of formal methods, particularly in hardware verification; instead of concentrating on correctness, the focus has turned more to bug finding.

For a discussion on the benefits of BMC in an industrial setting, we point the reader to a paper by Copty et al. from 2001, discussing experiments with BMC conducted at Intel in Haifa (Copty et al., 2001). In the same session at CAV, Bjesse et al. reported on bug-finding in the memory subsystem of an Alpha microprocessor (Bjesse et al., 2001). For some properties, SAT-based BMC reduced verification run-time from days to minutes on real, deep microprocessor bugs. Thus, by mid-2001, the SAT revolution was already well under way.

After the initial publication of the BMC idea, many optimizations and implementation techniques have been developed to improve on the original method. To name a few: tightly integrate the iterations with larger and larger n with an *incremental* SAT-solver in order to be able to reuse work between iterations (Een and Sörensson, 2003); use *reparameterization* in order to reduce the size of formulas that are generated for large n (Chauhan et al., 2004). The 2003 journal paper on advances in BMC is a good place to start for those who want to explore the technical ideas behind BMC further (Biere et al., 2003).

Nowadays, BMC is a key component in any industrial formal hardware verification set-up. Kunz' invited talk at FMCAD'07 illustrates this point (Kunz, 2007).

4 Temporal Induction

We have seen the use of Bounded Model Checking in safety property checking. What if, instead, we wish to prove that a property holds in *all* reachable states of a transition system; without the restriction to searching for counter-examples of length n or shorter? One option is, of course, to use the familiar BDD-based (unbounded) symbolic model checking (Clarke et al., 2000), but here we wish, again, to explore the use of SAT. We must therefore find a way to encode the problem directly as SAT instances, without using quantifiers.

Let us first introduce some notation. The symbol T^k stands for a "chain" of k transition relations:

$$T^k(s_0, \cdots, s_k) := T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k)$$

Now, consider the sequence of formulas $Base_0, Base_1, \ldots$, defined as follows:

$$Base_k \;:=\; I(s_0) \wedge T^k(s_0, \cdots, s_k) \wedge \neg P(s_k)$$

$Base_k$ is satisfiable if there is a path of length k through the transition relation T from an initial state to a bad state, and it is *unsatisfiable (UNSAT)* if there is no such path.

Iterating through all $Base_i$, from $i = 0$ and upwards, and checking if they are SAT or UNSAT, roughly corresponds to BMC, and is a bug-finding algorithm. If some $Base_k$ is satisfiable, the satisfying assignment of values to the bit-vectors in the state variables s_0 to s_k will give a shortest path from an initial state to a bad state.

The big question is "At what stage can we safely stop and conclude that there can be no such bug?".

One way to provide an answer is by *induction*. Let us call a path that only consists of good states a *good path*. If we can show that each path of length k starting in the initial state is a good path (induction base case), and that each good path of length k starting anywhere can only be extended by transitions that lead to a good state (induction step), then, by induction, all paths of any length starting in the initial state must be good paths.

The base case of the induction for k has already been defined above. In order to define the step case, let us first introduce some notation for asserting that P holds in a sequence of states:

$$P^k(s_0, \cdots, s_k) \;:=\; P(s_0) \wedge P(s_1) \wedge \cdots \wedge P(s_k)$$

We can then define a step case formula as follows:

$$Step_k \;:=\; T^{k+1}(s_0, \cdots, s_{k+1}) \wedge P^k(s_0, \cdots, s_k) \wedge \neg P(s_{k+1})$$

If $Step_k$ is satisfiable for some k, then there is a good path of length k that can be extended by going to a bad state.

A simple first induction algorithm can now be defined as follows:

```
i=0
while True do {
    if Sat(Base_i)
        return False    % counter-example
    if Unsat(Step_i)
        return True
    i=i+1
}
```

If we can find an i for which the base case is satisfiable, then we have a counter-example. If we can find an i for which both base case and step case are unsatisfiable, we have shown the property to hold for all reachable states.

The above algortithm is simple, but it is not complete; there are cases where the property holds but where the above algorithm does not terminate, i.e. no induction proof is found. This happens when there are paths of arbitrary length that satisfy the induction step $Step_k$. These paths must necessarily lie outside of the reachable state space. However, since our state space is finite (by assumption), there cannot be paths like this of arbitrary length unless they contain a loop. And since we are really only interested in shortest paths, and thus loop-free paths, we may add to the induction step that all considered paths must be loop-free.

We thus define a formula that expresses loop-freeness ("uniqueness" of states):

$$U^k(s_0, \cdots, s_k) \;:=\; \bigwedge_{0 \le i < j \le k} (s_i \neq s_j)$$

And redefine the step case formula as follows:

$$Step_k := T^{k+1}(s_0, \cdots, s_{k+1}) \wedge U^{k+1}(s_0, \cdots, s_{k+1}) \wedge P^k(s_0, \cdots, s_k) \wedge \neg P(s_{k+1})$$

The presented induction algorithm using the above step formula is indeed sound and complete. The soundness and completeness of the algorithm are easily shown, see (Een and Sörensson, 2003; Sheeran et al., 2000).

However, to make temporal induction work in practice, one must carefully consider exactly what SAT instances to present to the SAT solver, and in particular how to deal with the possibly expensive requirement that the paths considered in the termination check be loop-free. These aspects of the algorithm, its implementation using an incremental SAT solver, and an experimental evaluation of several variants of it, are presented in reference (Een and Sörensson, 2003).

Another way of improving on this basic algorithm is to *strengthen* the property P, i.e. to find a stronger property P' that implies P, and prove P' instead. The advantage of doing so is that the k that is needed to prove a stronger property might be much smaller, and thus the formulas that we have to deal with become smaller. An early implementation of this idea (automatically finding an equivalence relation between points in a hardware circuit) was developed for a BDD-based induction-like algorithm by van Eijk (van Eijk, 1998), and later adapted to SAT-based induction (Bjesse and Claessen, 2000). Recently, some alternative new techniques have been developed for automatically strengthening the induction hypothesis (Bradley and Manna, 2007; Case et al., 2007).

5 Supervisory Control

In control applications a computer controller is interacting with a physical plant. To analyse the behaviour of control applications it is thus necessary to have a model of both the control function and the physical plant, since the behaviour of the plant influences what the controller will do. *Supervisory Control Theory* (SCT) (Cassandras and Lafortune, 2008; Ramadge and Wonham, 1987, 1989) is an attempt to formalize modelling and reasoning about discrete-event control applications. In the SCT framework it is possible not only to verify if the closed-loop system satisfies given specifications, but also to synthesize a control function such that the given specifications are fulfilled. Typically, the models of the physical plant, the control functions and the specifications are given as finite state automata. However, the analysis of SCT problems suffers from state-space explosion, making large problems intractable. In this section, we will discuss how to formulate some SCT problems as SAT problems. This is intended as a first step towards enabling the supervisory control community to take advantage of the progress made within the computer science community on efficient SAT-solving.

SCT applies formal reasoning on a model of the uncontrolled process, the *plant*, and a model of the desired behaviour of the controlled system, the *specification*. From the plant and the specification, a safety device, called a *supervisor*, can be automatically synthesized. The supervisor controls the plant so that it always stays within the limits set by the specification, by dynamically disallowing the generation of events that might otherwise have given rise to behaviour outside the specification.

SCT proves that given a plant and a specification there will always exist an optimal supervisor guaranteeing that the specification will not be violated, while at the same time allowing the system to always fulfil its defined (sub-)tasks. Optimality here concerns restricting the given plant as little as possible. Such a supervisor is said to be *maximally permissive*, since it allows the controlled system the largest possible amount of freedom, in terms of event-generation, within the constraints set by the plant and the specification.

The control theoretic contribution, in the SCT framework, concerns the inclusion of a certain type of "controllability". The supervisor is mainly a safety device that prevents the plant from executing events that would take the controlled system outside the specified behaviour. However, not all events can be prevented from occurring; some events are *uncontrollable*, and the supervisor must never (try to) disable any of the uncontrollable events. It is known, (Ramadge and Wonham, 1987), that for a given specification and plant, a supervisor that guarantees that the entire specification can be achieved exists if and only if the specification is *controllable* with respect to the plant. This means that the specification must be such that it can be enforced without having to (try to) disable any uncontrollable events. If the original specification is not controllable with respect to the plant a controllable sub-behaviour of the specification has to be computed. It is known that the union of all controllable sub-behaviours of a specification with respect to a plant is also controllable thus a *supremal controllable sublanguage* exists. The supervisors task is to restrict the behaviour of the plant such that the supremal controllable sublanguage is achieved. If no controllable sublanguage exists which implies that the supremal controllable sublanguage is empty, then no supervisor exists.

In addition to controllability, which is a safety property, it is desired for the supervisor to be *nonblocking*. This is a progress property, enforced by the supervisor, that guarantees that at least one *marked* state is reachable from any state that it allows the controlled system to reach. Marked states typically represent (sub-)tasks that the system must always be able to finish. The condition that a state is nonblocking cannot be expressed as a property of the state alone, without considering possible progress from the state. However, *deadlocks*, non-marked states from which no transitions are possible, can be expressed as a property of each state, without considering possible progress. In this paper we focus on controllability and freedom from deadlocks. Thus we do not discuss the full supervisory control problem but instead focus on an important subclass of problems.

Although the SCT has traditionally focused on synthesis of supervisors, verification is a natural step within synthesis. Synthesis can be viewed as a series of verification tasks, where the process model (the plant) allows an automatic alteration of the suggested, and negatively verified, supervisor candidate. In this respect, the original specification can be viewed as a first supervisor candidate; if it is verified to be correct (controllable and nonblocking) then no further processing is necessary. Thus, by construction, a synthesized supervisor will always be verified to be correct.

This section starts by introducing the modelling formalism used in SCT. Then a manufacturing example is introduced and it is shown how to build a SAT-based model that can then be analysed using SCT inspired algorithms. The example is also used to illustrate in detail how to apply the SAT techniques discussed in the previous sections.

5.1 Modelling formalism

As a modelling formalism for supervisory control theory problems, *finite state automata* may be used. Formally, a finite state automaton (*FSA*), denoted by A, is defined as a four-tuple (Cassandras and Lafortune, 2008):

$$A = (Q, \Sigma, f, q_0)\,,$$

where Q is the finite set of *states*; Σ is the finite set of *events*, i.e. the alphabet, associated with the transitions in A; $f : Q \times \Sigma \to Q$ is the partial *transition function*: $f(q, \sigma) = p$ means that there is a transition labelled by event σ from state p to state q; $q_0 \in Q$ is the *initial* state.

The transition function, f, is partial and thus not all events are defined from all states. The *active event function* $\Gamma(q)$ denotes the set of all events σ for which $f(q, \sigma)$ is defined and is called the *active event set*. If $\sigma \in \Gamma(q)$ we say that the event σ is *enabled* in state q. The active event function is implicitly defined from the transition function f.

Typically, a model of the plant or the specification consists of different submodels focusing on different aspects. A specific composition operator *parallel composition*, see for example (Cassandras and Lafortune, 2008), is often used to compose a full model (plant or specification) from multiple submodels. This process is known as full synchronization.

Parallel composition, or full synchronization, of two automata states that for the event σ to be enabled in the composed system, each automaton that has σ in its alphabet also has to be in a state that contains σ in the active events set.

Let $A_1 = \left(Q^1, \Sigma^1, f^1, q_0^1\right)$ and $A_2 = \left(Q^2, \Sigma^2, f^2, q_0^2\right)$. The parallel composition of A_1 and A_2 is the automaton

$$A_1 || A_2 = \left(Q^1 \times Q^2, \Sigma^1 \cup \Sigma^2, f^{1||2}, (q_0^1, q_0^2)\right).$$

The transition function, $f^{1||2}$, is defined as

$$f^{1||2}((q^1, q^2), \sigma) := \begin{cases} (f^1(q^1, \sigma), f^2(q^2, \sigma)) & \text{if } \sigma \in \Gamma^1(q^1) \cap \Gamma^2(q^2) \\ (f^1(q^1, \sigma), q^2) & \text{if } \sigma \in \Gamma^1(q^1) \backslash \Sigma^2 \\ (q^1, f^2(q^2, \sigma)) & \text{if } \sigma \in \Gamma^2(q^2) \backslash \Sigma^1 \\ \textit{undefined} & \textit{otherwise.} \end{cases}$$

$\Gamma^{1||2}$ follows from the definition of $f^{1||2}$ and is given by

$$\Gamma^{1||2}(q^1, q^2) = \left(\Gamma^1(q^1) \cap \Gamma^2(q^2)\right) \cup \left(\Gamma^1(q^1) \backslash \Sigma^2\right) \cup \left(\Gamma^2(q^2) \backslash \Sigma^1\right).$$

Only reachable states are of importance in analyses; thus it is common to keep only the reachable subset of $Q^1 \times Q^2$ in the composition. The parallel composition operator is associative and commutative, and can thus be extended in a straightforward way to compose an arbitrary number of automata.

The transition function is written in infix notation; for example, $q \xrightarrow{\sigma} p$ denotes a transition from the state q to the state p associated with the event σ. This notation is further extended to strings in Σ^* in the natural way.

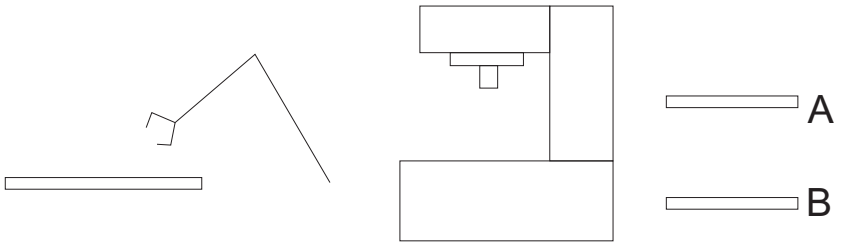
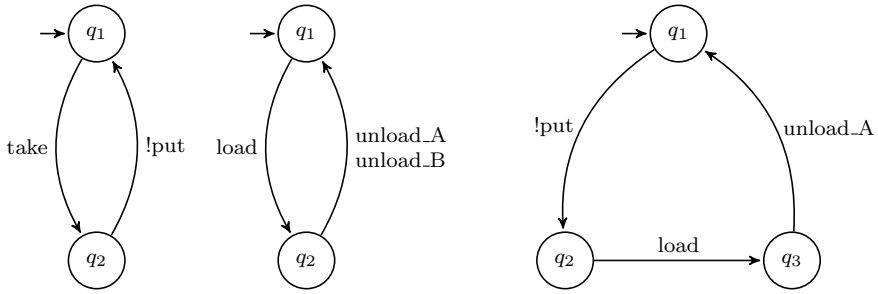


Fig. 1: A robot and a machine. The robot takes parts from the input buffer and puts them on the machine. The machine loads the part brought by the robot, and after processing unloads it to the output buffer A or B.



(a) Robot (b) Machine (c) Specification

Fig. 2: Automata models of the plant, consisting of the robot and the machine, and the specification. The exclamation mark (!) before an event name indicates that the event is uncontrollable. The alphabets are as follows; $\Sigma^{Robot} = \{take, !put\}$, $\Sigma^{Machine} = \{load, unload\_A, unload\_B\}$, and is $\Sigma^{Spec} = \{!put, load, unload\_A, unload\_B\}$. Note that the specification has no transition labelled *unload_B*, but this event is in the alphabet of the specification thus it is never allowed by the specification.

Example 2 (Modelling a robot and a machine using finite state automata)

As an example, consider one robot and one machine as shown in Fig. 1. The plant consists of the two automata shown in Fig. 2a and 2b that model the robot and the machine. It is assumed that the robot will spontaneously leave the part (i.e., corresponding to the put-event) in the machine after it has picked it up. Thus, the put-event is uncontrollable. Uncontrollable events are prefixed with an exclamation mark (!). We would like the system to fulfil the following specification (Fig. 2c): after each *!put* event there should follow event *load* followed by *unload_A*. This guarantees that the robot will not put a new part into the machine before the machine has consumed the current part. It also restricts the machine to use output buffer A only. In this example, the plant, P, is given by $Robot||Machine$ and the specification S consists of a single automaton. In general, both the plant and specification consist of multiple sub-plants and sub-specifications.

5.1.1 Controllability

In this section we will formulate controllability in the form of a propositional formula. Let P and S be two automata. Let Σ_u be the set of uncontrollable events and Σ^S be the alphabet of S. A state $(q^P, q^S) \in Q^P \times Q^S$ in the synchronized automaton $P||S$ is controllable if the following statement holds:

$$\Sigma^S \cap \Sigma_u \cap \Gamma(q^P) \subseteq \Gamma(q^S).$$

Uncontrollable states are the states in $P||S$ where P allows an uncontrollable event, but S disables the same event (by having it in its alphabet but not having it in the active event set of the current state).

Let P be the plant and S be a specification, and Σ_u be the set of uncontrollable events. S is controllable with respect to P and Σ_u if all reachable states of $P||S$ are controllable.

It is possible to transform the controllability problem to a problem of reachability of a forbidden state. To verify that specification $S = S_1|| \dots ||S_m$ is controllable with respect to the plant $P = P_1|| \dots ||P_n$, we will add an extra forbidden state to each sub-specification. The property for verification is that these new forbidden states should never be reachable. This is done by applying the following algorithm to all automata modelling specifications.

For every specification $S_i \in \{S_1, \dots S_m\}$ we will add a *forbidden* state q_f^i that will be reachable if and only if the system is uncontrollable. For every uncontrollable event $\sigma \in \Sigma_u$, and for every specification $S_i \in \{S_1, \dots S_m\}$ that contains this event in its alphabet ($\sigma \in \Sigma^i$), we will add transitions with the label σ from every state $q \in Q^i$ that does not have this event in its active event set ($\sigma \notin \Gamma^i(q)$) to the new forbidden state ($q \xrightarrow{\sigma} q_f^i$). The new sub-specifications are thus given by:

$$\begin{aligned} S^i_{updated} = \big(&Q^i \cup \{q_f^i\}, \\ &\Sigma^i, \\ &f^i \cup \{q \xrightarrow{\sigma} q_f^i, \sigma \in \Sigma^u \cap \Sigma^i, q \in Q^i \text{ s.t. } \sigma \notin \Gamma^i(q)\}, \\ &q_0^i\big) \end{aligned}$$

Applying the algorithm to the specification in Fig. 2c results in the automaton in Fig. 3. Note, that if the forbidden state q_f is reachable in $P||S$, then S is uncontrollable with respect to P.

5.1.2 Deadlocks

A state $q \in Q$ is a deadlock state if

$$\Gamma(q) = \emptyset,$$

that is if there are no transitions leaving the state. In many applications it is desirable for the system to not have any deadlocks, that is it should always be possible to continue. However, this is not always the case. If the system for example models a manufacturing system that has to produce five parts, then after those five parts are produced the

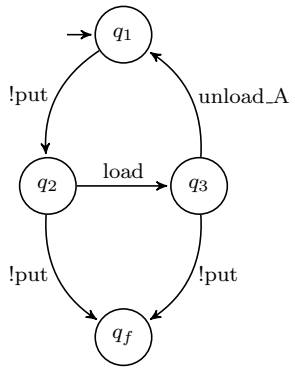


Fig. 3: The specification in Fig. 2c after extending it to include a forbidden state, q_f, which is reachable in the synchronized system as soon as the specification is not controllable with respect to the plant.

system is finished and has nothing else to do, thus leading to a desired deadlock. But, this problem can often be avoided by adding the possibility to return back to the initial state after finishing a set of tasks, or by adding self-loops at the desired final states. Thus, if a closed-loop system has deadlocks, it is usually due to some bad behaviour and the system needs to be modified to avoid deadlock states.

5.2 Encoding transition functions

We will now discuss how to encode a set of automata modelling the plant and the specification as a SAT-problem. Let $P = P_1||\dots||P_n$ be a plant and $S = S_1||\dots||S_m$ be a specification. Each automaton $A \in \{P_1, ...P_n, S_1...S_m\}$ can be encoded into a formula individually. The formula for the synchronous composition of automata is a conjunction of the formulas for the individual automata.

At least two approaches to encoding the state of an automaton are possible. The first approach, *one-hot* encoding, assigns one binary variable to each state. The binary variable is used to indicate if the state is active or not. Thus, the number of binary variables is equal to the number of states. The second approach assigns a unique integer in the interval from 0 to the $|Q| - 1$ to each state, where $|Q|$ is the number of states. In this case $\lceil \log_2 |Q| \rceil$ binary variables are needed to encode the states. The choice of encoding can have a great effect on the performance of the SAT-solver on the resulting SAT instances. However, the analysis of the performance of different encodings is non-trivial and is outside of the scope of this tutorial paper.

When one-hot encoding is used, the formula to model the initial state will be simply a conjunction stating that the state q_0^A of the automaton A is active, and all other states $Q^A \setminus \{q_0^A\}$ are inactive:

$$I^A(s) = s_{q_0^A} \wedge \bigwedge_{q \in Q^A \setminus \{q_0^A\}} \neg s_q$$

where vector s contains one binary variable for each $q \in Q^A$ of each automaton A, and s_q encodes if the state q is active or not.

The transition function should encode that state q will be active at the next step in two cases: (i) either state q is active at the current step and the event at the current step does not belong to the active events of the state, or (ii) among all transitions with state q as a destination state there is one whose source state and event are active at the current step.

Whether a state is active or not at the current step is encoded in the state vector s, while for events a vector e of *input variables* is used. For each event in the system, this vector has a binary variable that indicates that the event is active at the current step. For all input variables, the solver will try all possible values on each step while looking for a trace that violates a property. However, for the supervisory control problem, not all combinations of input variables are valid: only one event can be active in each step, and for each automaton this event should either belong to the set of active events of the currently active step, or lie outside of the automaton's alphabet. It is possible to use one binary variable for each automaton to specify that this *invariant* has never been broken, and add such a condition to the properties that are to be verified.

Taking all this into consideration, a complete formula for the transitions for an automaton A can be constructed as follows:

$$\begin{aligned} T^A(s, s') = & \bigwedge_{q \in Q^A} s'_q \leftrightarrow \Big(stay^A(q) \vee come^A(q) \Big) \\ & \wedge \Big(s'_{inv^A} \leftrightarrow \Big(s_{inv^A} \wedge (enabled^A \vee notInSigma^A) \Big) \Big) \end{aligned}$$

where

$$stay^A(q) = s_q \wedge \neg \left(\bigvee_{\sigma \in \Gamma^A(q)} e_\sigma \right)$$

$$come^A(q) = \bigvee_{p \in Q^A, \sigma \in \Sigma^A, f^A(p,e)=q} s_p \wedge e_\sigma$$

$$enabled^A = \bigwedge_{q \in Q^A} \left(q \rightarrow \bigvee_{\sigma \in \Gamma^A(q)} \left(e_\sigma \wedge \bigwedge_{\phi \in \Sigma^A \setminus \{\sigma\}} \neg e_\phi \right) \right)$$

$$notInSigma^A = \bigwedge_{\sigma \in \Sigma^A} \neg e_\sigma$$

where $a \leftrightarrow b$ is a short form of $((a \wedge b) \vee (\neg a \wedge \neg b))$.

Example 3 (Formula for the machine, robot and specification)

For our example of the robot and the machine, the state vector s will contain the binary variables $s_{q_1^R}$, $s_{q_2^R}$ and s_{inv^R} for the automaton that represents the robot, $s_{q_1^M}$, $s_{q_2^M}$ and s_{inv^M} for the automaton that represents the machine, and $s_{q_1^S}$, $s_{q_2^S}$, $s_{q_3^S}$, $s_{q_f^S}$ and s_{inv^S} for the specification automaton. The vector of input variables of the system will be $e = (e_{take}, e_{put}, e_{load}, e_{unload\_A}, e_{unload\_B})$.

The formula $I(s)$ that encodes the initial state of each automaton will be as follows:

$$I^R(s) = s_{q_1^R} \wedge \neg s_{q_2^R} \wedge s_{inv^R}$$

$$I^M(s) = s_{q_1^M} \wedge \neg s_{q_2^M} \wedge s_{inv^M}$$

$$I^S(s) = s_{q_1^S} \wedge \neg s_{q_2^S} \wedge \neg s_{q_3^S} \wedge \neg s_{q_f^S} \wedge s_{inv^S}$$

The first line says that the robot automaton is in state q_1 and not in state q_2, and that the invariant for the robot holds.

The formula to encode the transitions of the robot automaton says that state q_1 should be active if the automaton is already in that state and the event at the step is not *take*, or that the automaton was in the state q_2 and comes to q_1 on occurrence of event *put*. There is a similar expression for the other state of the automaton, and the invariant is also encoded.

$$
\begin{aligned}
T^R(s,s') = &\left(s'_{q_1^R} \leftrightarrow \left((s_{q_1^R} \wedge \neg e_{take}) \vee (s_{q_2^R} \wedge e_{put})\right)\right) \\
&\wedge \left(s'_{q_2^R} \leftrightarrow \left((s_{q_2^R} \wedge \neg e_{put}) \vee (s_{q_1^R} \wedge e_{take})\right)\right) \\
&\wedge \left(s'_{inv^R} \leftrightarrow \left(s_{inv^R} \wedge (enabled^R \vee notInSigma^R)\right)\right) \\
enabled^R = &\left(s_{q_1^R} \rightarrow (e_{take} \wedge \neg e_{put})\right) \wedge \left(s_{q_2^R} \rightarrow (e_{put} \wedge \neg e_{take})\right) \\
notInSigma^R = &\neg e_{take} \wedge \neg e_{put}
\end{aligned}
$$

The formula to encode the transitions of the machine is constructed in a similar way:

$$
\begin{aligned}
T^M(s,s') = &\left(s'_{q_1^M} \leftrightarrow \left((s_{q_1^M} \wedge \neg e_{load}) \vee (s_{q_2^M} \wedge e_{unload\_A}) \vee (s_{q_2^M} \wedge e_{unload\_B})\right)\right) \\
&\wedge \left(s'_{q_2^M} \leftrightarrow \left((s_{q_2^M} \wedge \neg e_{unload\_A} \wedge \neg e_{unload\_B}) \vee (s_{q_1^M} \wedge e_{load})\right)\right) \\
&\wedge \left(s'_{inv^M} \leftrightarrow \left(s_{inv^M} \wedge (e^M_{enabled} \vee e^M_{notInSigma})\right)\right) \\
enabled^M = &\left(s_{q_1^M} \rightarrow (e_{load} \wedge \neg e_{unload\_A} \wedge \neg e_{unload\_B})\right) \\
&\wedge (s_{q_2^M} \rightarrow ((e_{unload\_A} \wedge \neg e_{load} \wedge \neg e_{unload\_B}) \\
&\quad \vee (e_{unload\_B} \wedge \neg e_{load} \wedge \neg e_{unload\_A}))) \\
notInSigma^M = &\neg e_{load} \wedge \neg e_{unload\_A} \wedge \neg e_{unload\_B}
\end{aligned}
$$

The formula to encode the transitions of the specification, see Figure 3, is constructed in the same manner:

$$
\begin{aligned}
T^S(s,s') = &\left(s'_{q_1^S} \leftrightarrow \left((s_{q_1^S} \wedge \neg e_{put}) \vee (s_{q_3^S} \wedge e_{unload\_A})\right)\right) \\
&\wedge \left(s'_{q_2^S} \leftrightarrow \left((s_{q_2^S} \wedge \neg e_{load} \wedge \neg e_{put}) \vee (s_{q_1^S} \wedge e_{put})\right)\right) \\
&\wedge \left(s'_{q_3^S} \leftrightarrow \left((s_{q_3^S} \wedge \neg e_{put} \wedge \neg e_{unload\_A}) \vee (s_{q_2^S} \wedge e_{load})\right)\right) \\
&\wedge \left(s'_{q_f^S} \leftrightarrow \left(s_{q_f^S} \vee (s_{q_2^S} \wedge e_{put}) \vee (s_{q_3^S} \wedge e_{put})\right)\right) \\
&\wedge \left(s'_{inv^S} \leftrightarrow \left(s_{inv^S} \wedge (enabled^S \vee notInSigma^S)\right)\right) \\
enabled^S = &\left(s_{q_1^S} \rightarrow (e_{put} \wedge \neg e_{load} \wedge \neg e_{unload\_A})\right) \\
&\wedge \left(s_{q_2^S} \rightarrow ((e_{load} \wedge \neg e_{put} \wedge \neg e_{unload\_A}) \vee (e_{put} \wedge \neg e_{load} \wedge \neg e_{unload\_A}))\right) \\
&\wedge \left(s_{q_3^S} \rightarrow ((e_{unload\_A} \wedge \neg e_{put} \wedge \neg e_{load}) \vee (e_{put} \wedge \neg e_{load} \wedge \neg e_{unload\_A}))\right) \\
&\wedge \left(s_{q_f^S} \rightarrow (\neg e_{put} \wedge \neg e_{load} \wedge \neg e_{unload\_A})\right) \\
notInSigma^S = &\neg e_{put} \wedge \neg e_{load} \wedge \neg e_{unload\_A}
\end{aligned}
$$

5.3 Verification

In the previous section we have shown how to encode the transition structure of all plants and specifications. However, the full specification is given not only by the transition structure of the specification automaton, but also by the property that the supervisor is controllable with respect to the plant. Controllability is an important property for the system because it models whether or not supervisor candidate tries to disable something it has no influence over, for example the !*put* event in Fig. 2. In this section we discuss how to verify if a specification is controllable with respect to a given plant, and also how to verify if a system is deadlock-free.

5.3.1 No uncontrollable states

A controllability problem is present if any of the forbidden states added to the specifications are reachable, which gives us the following property for each specification S:

$$P^S_{contr}(s) = \neg s_{q_f^S}$$

5.3.2 No deadlock states

Deadlock is a situation when no transition can be taken from some reachable state. This can happen when each event is disabled in at least one automaton that has the event in its alphabet, and is thus disabled in the composed system. For n plants and m specifications with the union alphabet

$$\Sigma = \bigcup_{A \in \{P_1,\ldots P_n, S_1,\ldots S_m\}} \Sigma^A$$

this can be expressed as follows:

$$P_{noDeadl}(s) = \neg \left(\bigwedge_{\sigma \in \Sigma} \bigvee_{A \in \{P_1, \ldots P_n, S_1, \ldots S_m\}} disabled^A(\sigma) \right)$$

$$disabled^A(\sigma) = \bigvee_{q \in Q^A, \sigma \in \Sigma^A \setminus \Gamma^A(q)} s_q$$

Example 4 (Defining properties to verify)

A property we would like to verify is if there are no uncontrollable or deadlock states. Since we are interested in feasible paths only, we will include an invariant in this property: the solver should look for traces where the property does not hold while the invariant still holds; if the invariant is broken, the property holds automatically. This can be formulated as follows:

$$P_1(s) = (P_{noUncontr}(s) \wedge P_{noDeadl}(s)) \vee \neg s_{inv^R} \vee \neg s_{inv^M} \vee \neg s_{inv^S}$$

$$P_{noUncontr}(s) = \neg s_{q_f^S}$$

$$\begin{aligned} P_{noDeadl}(s) = \neg \Big(& \left(s_{q_2^R} \right) && \% \textit{ take} \\ & \wedge \left(s_{q_1^R} \vee s_{q_f^S} \right) && \% \textit{ put} \\ & \wedge \left(s_{q_2^M} \vee s_{q_1^S} \vee s_{q_3^S} \vee s_{q_f^S} \right) && \% \textit{ load} \\ & \wedge \left(s_{q_1^M} \vee s_{q_1^S} \vee s_{q_2^S} \vee s_{q_f^S} \right) && \% \textit{ unload\_A} \\ & \wedge \left(s_{q_1^M} \right) \Big) && \% \textit{ unload\_B} \end{aligned}$$

5.3.3 Verification procedure

Verification will explore states of the synchronized system shown in Fig. 4, which is the parallel composition of the plants and the specification. We will follow the BMC procedure described in Section 3. We start with one step, and successively increase the number of steps until a counter-example is found or the diameter of the system is reached. The formula for one step will look as follows:

$$I^R(s_0) \wedge I^M(s_0) \wedge I^S(s_0) \wedge \neg P_1(s_0)$$

$P_1(s_0)$ is true if the specification holds, i.e. state s_0 is both controllable and deadlock-free. Thus, if it is possible to satisfy the above formula then the specification does not hold in the initial state. The bounded model checker will conclude that specification holds in the initial state, since the SAT-solver indicates that the above formula is not satisfiable. Adding one more step will give the following formula:

$$\begin{aligned} & I^R(s_0) \wedge I^M(s_0) \wedge I^S(s_0) \\ & \wedge T^R(s_0, s_1) \wedge T^M(s_0, s_1) \wedge T^S(s_0, s_1) \\ & \wedge (\neg P_1(s_0) \vee \neg P_1(s_1)) \end{aligned}$$

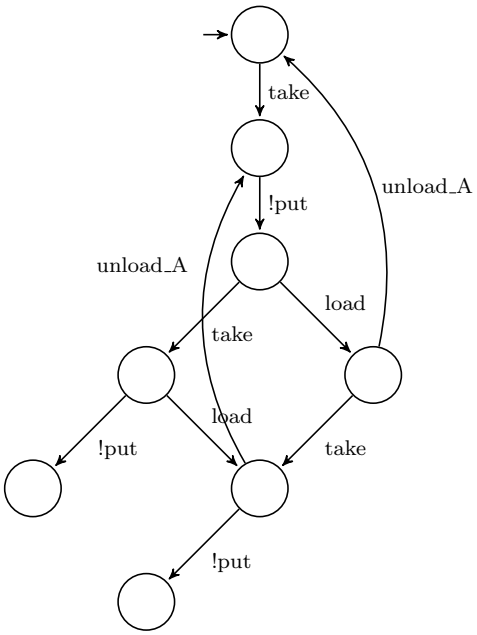


Fig. 4: $R||M||S$ for the plant models given in Fig. 2a and 2b and the specification in Fig. 3.

This formula is also unsatisfiable. Continuing this way until the fourth step, we obtain the formula

$$
\begin{aligned}
& I^R(s_0) \wedge I^M(s_0) \wedge I^S(s_0) \\
& \wedge T^R(s_0, s_1) \wedge T^M(s_0, s_1) \wedge T^S(s_0, s_1) \\
& \wedge T^R(s_1, s_2) \wedge T^M(s_1, s_2) \wedge T^S(s_1, s_2) \\
& \wedge T^R(s_2, s_3) \wedge T^M(s_2, s_3) \wedge T^S(s_2, s_3) \\
& \wedge T^R(s_3, s_4) \wedge T^M(s_3, s_4) \wedge T^S(s_3, s_4) \\
& \wedge (\neg P_1(s_0) \vee \neg P_1(s_1) \vee \neg P_1(s_2) \vee \neg P_1(s_3) \vee \neg P_1(s_4)),
\end{aligned}
$$

which is satisfiable. The solver returns as well an assignment of the variables that produces this counter-example, from which it is possible to extract the sequence of events that lead to the bad state:

$$take \rightarrow !put \rightarrow take \rightarrow !put.$$

The trace leads to an added forbidden state, which says that the state in the trace just before this last forbidden state is uncontrollable.

With a fixed number of steps it is only possible to reveal problems or verify their absence for that given number of steps. To verify that there are no problems in the system for any number of steps, it is possible to determine the diameter of the system, which could be difficult, and run the verification for that number of steps. It is also possible to use temporal induction as described in Section 4. We will omit the formula

for the base case and the induction steps here. However, we will assume, in what follows, that verifications are performed for the whole system, either by induction or by taking a number of steps big enough to cover all states of the system.

5.4 Synthesis via iterative specification refinement

Verification could reveal deadlocks and controllability problems, but with the help of a number of verifications, it is possible to add extra specifications to the system that would remove bad behaviour from the system. This can be seen as a form of incremental synthesis, although the result is not guaranteed to have the nonblocking property.

Each verification will either confirm that there are no bad states, or will give a counter-example in the form of a shortest trace (string) leading to such a state. Having a trace, it is possible to automatically add an extra specification in addition to the existing ones to forbid taking the transition that led to a bad state. For deadlocks the last transition of the trace is the transition that transfers the models to the deadlock state. For an uncontrollable state the trace has to be shortened by one, since the last transition led to an artificially added state q_f.

The specification for a trace $D = \sigma_1, \ldots, \sigma_n$ of n events will have $n+1$ states $q_1 \ldots q_n, q_{good}$, with q_1 being the initial state. The alphabet Σ of the specification will contain all events of the trace. For each step $i \in \{1 \ldots n-1\}$ of the trace there will be a transition $q_i \xrightarrow{\sigma_i} q_{i+1}$ with the trace event σ_i, and transitions from all other events to the "good" state $q_i \xrightarrow{\Sigma \setminus \{\sigma_i\}} q_{good}, i \in \{1 \ldots n\}$. There will be no transition leaving state q_n named σ_n. The state q_{good} will have transitions with all events to itself: $q_{good} \xrightarrow{\Sigma} q_{good}$. More formally, create a new specification that has the following structure and add to the existing set of specifications.

$$
\begin{aligned}
S_{extra} = \big(& (q_1, \ldots, q_n, q_{good}), && \% \text{ states } Q \\
& \{\sigma, \sigma \in D\}, && \% \text{ alphabet } \Sigma \\
& \{q_i \xrightarrow{\sigma_i} q_{i+1}, i = 1 \ldots n-1\} \\
& \quad \cup \{q_i \xrightarrow{\Sigma \setminus \{\sigma_i\}} q_{good}, i = 1 \ldots n\} \\
& \quad \cup \{q_{good} \xrightarrow{\Sigma} q_{good}\}, && \% \text{ transition function } f \\
& q_1 \big) && \% \text{ initial state}
\end{aligned}
$$

The new specification states that any transition that does not belong to the bad trace will lead to the good state, while the last step of the bad trace is disallowed.

Verification followed by adding an extra specification should be repeated until no more bad states are found.

Example 5 (Incrementally adding specifications)

The verification procedure reveals a shortest trace to a bad state: *take, !put, take, !put*. Since the trace leads to state q_f of the specification, it is necessary to avoid reaching the state that is one step before q_f. To ensure that, the specification shown in Fig. 5 that forbids the sequence *take, !put, take* could be added to the system.

Verification of a new set of automata will reveal one more trace, *take, !put, load, take, !put*, that shows one more controllability problem. By forbidding this path as well, finally we will get to a system that allows the repetition of only four events in the loop:

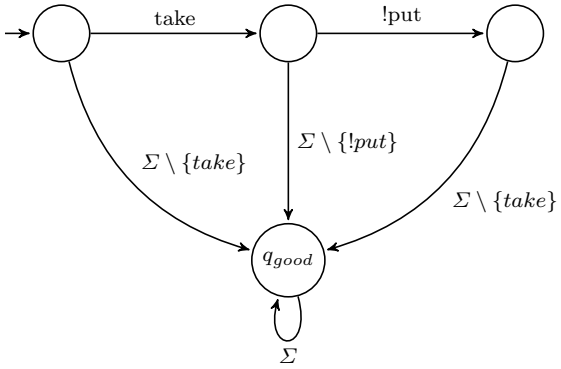


Fig. 5: Extra specification to avoid the first uncontrollable state reachable via the sequence *take* → *!put* → *take*.

take, !put, load, unload_A. The verification procedure for this system terminates with a proof that no bad state can be reached in the system. Thus, the set of the original specifications and the newly created specifications may be used as the supervisor. Note, that it might be possible for the initial state to be uncontrollable; in this case no non-empty supervisor exists. If this is the case then the system is not allowed to start. However, if a non-empty supervisor exists, the above procedure will compute it.

5.5 Discussion of the use of SAT in the supervisory control example

It is relatively easy to encode safety properties like controllability and deadlock avoidance as a propositional formula for use with bounded model checking. However, the nonblocking property cannot be expressed simply by a propositional formula. Apart from verification, it is also possible to implement an incremental synthesis procedure with the help of a number of verifications. In (Voronov and Åkesson, 2008) some experimental results are presented. In this paper we have shown one way to encode SCT related problems, but other encodings are possible as well. Which encoding to use in order to have the highest performance from the SAT-based tools is still an open issue.

6 Discussion and conclusion

We have very briefly catalogued the simulataneous development of modern SAT-solvers and their applications. Another popular model checking technique, based on interpolants, was introduced in 2003 by McMillan (McMillan, 2003). For a survey of SAT-based Formal Verification, see references (Amla et al., 2005; Prasad et al., 2005). Bryant and Kukula's 2002 survey paper on Formal Methods in Functional Verification (Bryant and Kukula, 2003) is a fascinating journey from the early attempts to use inefficient decision procedures up to the period just after the SAT revolution. It ends by cautioning that although the successes in industrial application are encouraging, improvements in speed and capacity of the basic engines are still needed. That is still true today, so that the new developments outlined in section 2.5 are eagerly awaited by the users of SAT-based tools.

It should also be noted that SAT is often mixed with other technologies (such as BDDs or dynamic (simulation-based) verification) in industrial-strength tools. IBM's SixthSense system for circuit verification is a good example of this development (Mony et al., 2004). Bentley's invited talk on microprocessor verification, given at the Computer Aided Verification conference in 2005, not only shows the enormity of the verification problems that we face but also points towards the use of SAT as a means to bridge dynamic (simulation-based) and formal verification (Bentley, 2005). For a good recent overview of the use of all of SAT, Bounded Model Checking and temporal induction, the reader is referred to FMCAD 2007 (Baumgartner and Sheeran, 2007).

As the complexity of the hardware and software systems that we wish to verify grows inexorably, it is increasingly clear that automated reasoning at a level of abstraction above the bit level is needed. This has led to a surge of interest in *Satisfiability Modulo Theories (SMT)* – the combination of SAT with additional theories such as linear arithmetic or bit-vectors, (Cimatti, 2008). The move upwards in level of abstraction is also reflected in increasing research activity in automated reasoning for Quantified Boolean Formulas and even First Order Logic.

In the area of Discrete Event Systems, we have taken as a tutorial example the use of SAT in supervisory control. We hope that this introduction to SAT-solving in practice will whet the appetite of researchers in new fields, outside our familiar area of Computer Aided Design of Electronic Circuits. Although much work remains to be done, we hope that this paper can stimulate work on the border between discrete event systems and SAT-solving. We look forward to fruitful collaboration between our research communities.

7 Acknowledgements

The authors gratefully acknowledge Joao Marques-Silva for useful comments on earlier drafts of this paper, and Bengt Lennartson and Martin Fabian from the Signals and Systems Department at Chalmers for taking the initiative to include a special session on SAT at the International Workshop on Discrete Event Systems, May, 2008.

References

The international SAT Competitions. http://www.satcompetition.org/.

Parosh Aziz Abdulla, Per Bjesse, and Niklas Een. Symbolic Reachability Analysis Based on SAT-Solvers. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS 2000*, pages 411–425, 2000. doi: 10.1007/3-540-46419-0_28.

Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P Kurshan, and Ken L McMillan. An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment. In Dominique Borrione and Wolfgang Paul, editors, *Correct Hardware Design and Verification Methods CHARME-2005*, pages 254–268, Saarbrücken, Germany, 2005. Springer. doi: 10.1007/11560548_20.

Jason Baumgartner and Mary Sheeran, editors. *Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*. IEEE Computer Society, 2007.

Bob Bentley. Validating a Modern Microprocessor, invited talk at CAV 2005, 2005.

Armin Biere. PicoSAT Version 535, System description for the SAT competition. Technical report, 2007a.

Armin Biere. Short History on SAT Solver Technology and What is Next?, invited talk, 2007b.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS99*, number 97, pages 193–207, Amsterdam, The Netherlands, 1999a. Springer. doi: 10.1007/3-540-49059-0_14.

Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. Verifying Safety Properties of a PowerPC. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification CAV-99*, number 97, pages 60–71, Trento, Italy, 1999b. Springer. doi: 10.1007/3-540-48683-6_8.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. *Advances in Computers*, 58:118–149, 2003. doi: 10.1016/S0065-2458(03)58003-2.

Per Bjesse and Koen Claessen. SAT-Based Verification without State Space Traversal. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design FMCAD 2000*, pages 372–389, Austin, Texas, 2000. Springer. doi: 10.1007/3-540-40922-X_23.

Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification, CAV'01*, pages 454–464, Paris, France, 2001. Springer-Verlag. ISBN 3-540-42345-1. doi: 10.1007/3-540-44585-4_44.

A. Borälv and Gunnar Stålmarck. Prover Technology in Railways. In *Industrial-Strength Formal Methods*. Academic Press, 1998.

Aaron R Bradley and Zohar Manna. Checking Safety by Inductive Generalization of Counterexamples to Induction. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 173–180, Austin, Texas, 2007. IEEE. ISBN 0-7695-3023-0. doi: 10.1109/FAMCAD.2007.15.

Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. ISSN 0018-9340. doi: 10.1109/TC.1986.1676819.

Randal E. Bryant and J. H. Kukula. Formal Methods for Functional Verification. In Andreas Kuehlmann, editor, *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*. Springer, 2003. ISBN 978-1-4020-7391-5.

Michael L Case, Alan Mishchenko, and Robert K Brayton. Automated Extraction of Inductive Invariants to Aid Model Checking. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 165–172, Austin, Texas, 2007. IEEE. ISBN 0-7695-3023-0. doi: 10.1109/FAMCAD.2007.12.

Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer US, Boston, MA, 2nd edition, 2008. ISBN 978-0-387-33332-8. doi: 10.1007/978-0-387-68612-7.

Pankaj Chauhan, Edmund M. Clarke, and Daniel Kroening. A SAT-based algorithm for reparameterization in symbolic simulation. In *Proceedings of the 41st annual conference on Design automation - DAC '04*, page 524, New York, New York, USA, 2004. ACM Press. ISBN 1581138288. doi: 10.1145/996566.996711.

Alessandro Cimatti. Beyond Boolean SAT: Satisfiability modulo theories. In *9th International Workshop on Discrete Event Systems WODES-2008*, pages 68–73, Göteborg, Sweden, 2008. IEEE. ISBN 978-1-4244-2592-1. doi: 10.1109/WODES.2008.4605924.

Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sörensson. SAT-solving in practice. In *9th International Workshop on Discrete Event Systems WODES-2008*, pages 61–67, Göteborg, Sweden, 2008. IEEE. ISBN 978-1-4244-2592-1. doi: 10.1109/WODES.2008.4605923.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000. ISBN 978-0-262-03270-4.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on*, pages 151–158. ACM Press, 1971. doi: 10.1145/800157.805047.

Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification, CAV'01*, pages 436–453, Paris, France, 2001. Springer-Verlag. ISBN 3-540-42345-1. doi: 10.1007/3-540-44585-4_43.

Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960. ISSN 00045411. doi: 10.1145/321033.321034.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. ISSN 0001-0782. doi: 10.1145/368273.368557.

Niklas Een. Symbolic Reachability Analysis based on SAT-Solvers (Master's Thesis). Technical report, Uppsala University, Uppsala, 1999.

Niklas Een. Practical SAT, invited tutorial at Int. Conf. on Formal Methods in Computer Aided Design, 2007.

Niklas Een and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Theory and Applications of Satisfiability Testing*, pages 61–75, 2005. doi: 10.1007/11499107_5.

Niklas Een and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science: Proceedings of First International Workshop on Bounded Model Checking*, 89(4):543–560, 2003. ISSN 15710661. doi: 10.1016/S1571-0661(05)82542-3.

Niklas Een and Niklas Sörensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, 2919:502–518, 2004. doi: 10.1007/978-3-540-24605-3_37.

Niklas Een and Niklas Sörensson. MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition, 2005.

Niklas Een, Alan Mishchenko, and Niklas Sörensson. Applying Logic Synthesis for Speeding Up SAT. In *Proceedings of the 10th international conference on Theory and applications of satisfiability testing*, pages 272–286. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-72788-0_26.

Evgeueni Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation and Test in Europe DATE 2002*, pages 142–149. IEEE Comput. Soc, 2002. ISBN 0-7695-1471-5. doi: 10.1109/DATE.2002.998262.

J.F. Groote, J.W.C. Koorn, and S.F.M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd (Extended abstract). In *10th Annual Conference on Computer Assurance (COMPASS'95)*, Gaithersburg, Maryland, 1995.

Jinbo Huang. The Effect of Restarts on the Efficiency of Clause Learning. In *IJCAI 2007*. AAAI Press, 2007.

Henry Kautz and Bart Selman. Planning as satisfiability. In *ECAI 1992*, pages 359–363. John Wiley & Sons, Inc., 1992. ISBN 0471936081.

W. Kunz. Formal Verification of Systems-on-Chip – Industrial Experiences and Scientific Perspectives. Invited talk at FMCAD-07, 2007.

W. Kunz and D.K. Pradhan. Recursive learning: a new implication technique for efficient solutions to CAD problems: Test, verification, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(9): 1143–1158, 1994. ISSN 02780070. doi: 10.1109/43.310903.

Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, 1992. ISSN 02780070. doi: 10.1109/43.108614.

Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993. ISSN 00200190. doi: 10.1016/0020-0190(93)90029-9.

Joao P. Marques-Silva. Practical applications of Boolean Satisfiability. In *9th International Workshop on Discrete Event Systems WODES-2008*, pages 74–80, Göteborg, Sweden, 2008. IEEE. ISBN 978-1-4244-2592-1. doi: 10.1109/WODES.2008.4605925.

Joao P. Marques-Silva and Karem A. Sakallah. GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, San Jose, California, 1996. IEEE Comput. Soc. Press. ISBN 0-8186-7597-7. doi: 10.1109/ICCAD.1996.569607.

Joao P. Marques-Silva and Karem A. Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 152–161. IEEE Comput. Soc, 1997. ISBN 0-8186-7831-3. doi: 10.1109/FTCS.1997.614088.

Ken L McMillan. Interpolation and SAT-Based Model Checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification CAV-03*, pages 1–13, Boulder, CO, 2003. Springer. doi: 10.1007/978-3-540-45069-6_1.

Hari Mony, Jason Baumgartner, Viresh Paruthi, Robert Kanzelman, and Andreas Kuehlmann. Scalable Automated Verification via Expert-System Guided Transformations. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design FMCAD-2004*, pages 159–173, Austin, Texas, 2004. Springer. doi: 10.1007/978-3-540-30494-4_12.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001. ISBN 1581132972. doi: 10.1145/378239.379017.

Gi-Joon Nam, Karem A. Sakallah, and Rob A. Rutenbar. Satisfiability-based layout revisited. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays - FPGA '99*, pages 167–175, New York, New York, USA, 1999. ACM Press. ISBN 1581130880. doi: 10.1145/296399.296450.

Knot Pipatsrisawat and Adnan Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In Joao Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing SAT 2007*, pages 294–299, Lisbon, Portugal, 2007. Springer. doi: 10.1007/978-3-540-72788-0_28.

Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for*

Technology Transfer (STTT), 7(2):156–173, 2005. ISSN 1433-2779. doi: 10.1007/s10009-004-0183-4.

Peter J.G. Ramadge and W. Murray Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization*, 25(1):206, 1987. ISSN 03630129. doi: 10.1137/0325013.

Peter J.G. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989. doi: 10.1109/5.21072.

M. Säflund. Modelling and formally verifying systems and software in industrial applications. In Fu-jung Hshu and Xu Furong, editors, *Proceedings of the Second International Conference on Reliability, Maintainability and Safety (ICRMS '94)*, Beijing, China, 1994.

Ohad Shacham and Emmanuel Zarpas. Tuning the VSIDS decision heuristic for bounded model checking. In *4th International Workshop on Microprocessor Test and Verification - Common Challenges and Solutions*, pages 75–79. IEEE, 2003. ISBN 0-7695-2045-6. doi: 10.1109/MTV.2003.1250266.

Mary Sheeran and Gunnar Stålmarck. A Tutorial on Stålmarcks Proof Procedure for Propositional Logic. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design*, pages 82–99, Palo Alto, CA, 1998. Springer. doi: 10.1007/3-540-49519-3_7.

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design*, pages 127–144. 2000. doi: 10.1007/3-540-40922-X_8.

L.G. Silva, Joao P. Marques-Silva, L.M. Silveira, and Karem A. Sakallah. Timing analysis using propositional satisfiability. In *1998 IEEE International Conference on Electronics, Circuits and Systems. Surfing the Waves of Science and Technology*, volume 3, pages 95–98. IEEE, 1998. ISBN 0-7803-5008-1. doi: 10.1109/ICECS.1998.813943.

Mandayam Srivas and Albert Camilleri, editors. *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg, 1996. ISBN 3-540-61937-2. doi: 10.1007/BFb0031795.

Gunnar Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula, 1990.

Gregory S. Tseitin. On the complexity of derivation in propositional calculus. In A.O. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian)*, pages 234–259. Steklov Mathematical Institute, 1968.

C.A.J. van Eijk. Sequential equivalence checking without state space traversal. In *Proceedings Design, Automation and Test in Europe DATE-98*, pages 618–623, Paris, France, 1998. IEEE Comput. Soc. ISBN 0-8186-8359-7. doi: 10.1109/DATE.1998.655922.

Alexey Voronov and Knut Åkesson. Supervisory control using satisfiability solvers. In *9th International Workshop on Discrete Event Systems WODES-2008*, pages 81–86. IEEE, 2008. ISBN 9781424425938. doi: 10.1109/WODES.2008.4605926.

Hantao Zhang. SATO: An Efficient Propositional Prover. In William McCune, editor, *14th International Conference on Automated Deduction*, volume 2, pages 272–275, Townsville, North Queensland, Australia, 1996. Springer. doi: 10.1007/3-540-63104-6_28.