Addressing the Performance Bottlenecks of Device Driver Synthesis

Alexander Legg

Supervised by
Leonid Ryzhyk, Nina Narodytska, and Gernot Heiser

June 20, 2016

Contents

Abstract

Device drivers are notorious for their contribution to the failure of software systems. A driver forms the interface between two complicated systems: an operating system and a hardware device. As a result, driver development is complicated and highly prone to human error. Driver reliability has been the focus of much research over the past two decades. One result of this research is device driver synthesis.

Device driver synthesis aims to remove human error from the development process by removing humans. A synthesis tool can take a

formal specification of requirements and produce a system. Thus, a driver can be synthesised by providing the tool with a specification of the OS and the hardware that it must interface with. The resulting driver is guaranteed to be correct with respect to the specifications.

The drawback of this approach is that synthesis is a computationally intensive task belonging to 2-EXPTIME. In practice synthesis tools are capable of producing drivers though there remain cases where synthesis is impossible. In this thesis I will present an algorithm that provides results on many of these previously unattainable cases.

Acknowledgements

Acknowledge some people

1 Introduction

This is the introduction.

1.1 Device Drivers

Device drivers are the software that allows the operating system to interface with hardware. The role of the driver is to manipulate the inputs of the device so that it remains in a error-free state and correctly handles the requests of the operating system. By way of example consider an ethernet driver. The driver accepts requests to send and receive data packets from the OS and acts on those requests by reading from and writing to buffers on the device. It must ensure that those buffers are maintained in a usable state by correctly updating a register containing the location of the head of the buffer queue.

According to a study performed in 2011 [Palix et al., 2011], drivers account for approximately 57% of the lines of code in the Linux kernel and subsequently is the largest source of bugs. The study also analysed the staging directory of the kernel, which contains all in-progress drivers, and found it to contain the highest fault rate (faults per line of code) out of any directory in the kernel. The results of the study give evidence to the widely held belief that correct drivers are hard to produce.

Consequences of buggy drivers

This thesis focuses on automatic construction of correct drivers as a solution to the driver problem. Alternate approaches, of which there are many, will be discussed in Chapter 3.

1.2 Synthesis

Explain what synthesis is

- Briefly describe the current state of the art
- Explain state space explosion

1.3 Synthesis with SAT

Explain SAT solvers

- Explain how they might help with state space explosion
- (Explain that there is room for both approaches in the world)
- Explain QBF and/or why this is nontrivial

2 Background

Synthesis is a process that demands mathematical formalisation in order to provide a strong guarantee of the correctness of the resultant software. As such we require a mathematical language to describe the system we wish to produce, the environment in which it operates, and the properties we want the system to adhere to. This chapter will outline that language and the ways we can reason about what we describe in it.

2.1 Temporal Logic

Devices drivers are an example of a reactive system. A reactive system is a system that is in a continuous process of responding to input from its environment. A driver accepts requests from the operating system and state information from the device, and it responds by sending commands to the device and reporting to the operating system.

In an ordinary program, correctness can be ascertained by observing the final output. A reactive system does not terminate and is only considered correct if it adheres to its specification indefinitely. A formalism of a reactive system must then consider the concept of time in order to specify its correctness property.

2.1.1 Kripke Structures

A reactive system can be thought of as a sequence of *states*. The system *transitions* between these states as it responds to its inputs. A Kripke structure [Kripke, 1963] formalises this notion and provides us will the language to reason about a reactive system.

A Kripe structure M is defined by the tuple $M = (S, S_0, R, L)$ with respect to a set of atomic propositions AP.

- A finite set of states, S,
- a subset of initial stats $S_0 \subseteq S$,
- a transition relation $R \subseteq S \times S$, and
- a labelling function $L : S \to 2^{AP}$.

The transition relation defines how the system moves between states. It must be left-total, i.e. for every $s \in S$ there is an $s' \in S$ s.t. $R(s, s')$. The labelling function maps every state in S to a set of atomic propositions that hold in that state of the system.

We often consider *paths* or *runs* of a Kripke structure. A path is a sequence of states $\pi = s_0, s_1, s_2, ...$ such that $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

2.1.2 Linear Temporal Logic

Kripke structures lay the groundwork for reasoning about reactive systems. Using the labelling function we may define desirable properties for the system. What is now lacking is a means of bringing states together to reason about the system as a whole. This requires a logical language that can express temporal properties.

Temporal logic takes propositional logic and provides additional semantics for the concept of time. In a Kripke structure this refers to the expressiveness to reason about runs of the system. This allows us to define properties that must be true for the entire execution of a reactive system.

Linear temporal logic (LTL) allows for statements that refer to a single run of a Kripke structure. Pnueli introduced LTL in 1977 [Pnueli, 1977] to succinctly describe the outcomes of program execution by referring to global invariants and eventualities. The syntax is:

- ϕ is a propositional formula referring to the current state,
- $X\phi$ - ϕ is true in the next state of the execution,
- $F\phi$ - Eventually (finally) ϕ will be true, and
- $G\phi$ - ϕ is always (globally) true.

- $\phi_1 U \phi_2$ - ϕ_1 holds until ϕ_2 holds.

These operators are semantically defined with respect to a Kripke structure $M = (S, S_0, R, L)$. We use $M, s \models \phi$ to denote ϕ holds true at state $s \in S$ of structure M. We definte $\models$ recursively:

- $M, s \models \phi$ iff $\phi \in L(s)$.
- $M, s \models \neg\phi$ iff not $(M, s \models \phi)$.
- $M, s \models \phi_1 \wedge \phi_2$ iff $(M, s \models \phi_1) \wedge (M, s \models \phi_2)$.
- $M, s \models \phi_1 \vee \phi_2$ iff $(M, s \models \phi_1) \vee (M, s \models \phi_2)$.
- $M, s \models X\phi$ iff for some state s', $R(s, s') \wedge M, s' \models \phi$.
- $M, s_0 \models F\phi$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi))$.
- $M, s_0 \models G\phi$ iff for some path $(s_0, s_1, ...)$, $\forall i(i \geq 0 \wedge (M, s_i \models \phi))$.
- $M, s_0 \models \phi_1 U \phi_2$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.

Throughout this thesis will use F and G to represent the *finally* and *globally* operators. Elsewhere in the literature $\Diamond$ and $\Box$ are sometimes used to represent the same.

2.1.3 Computation Tree Logic

In addition to LTL, which is used to formalise properties about a single execution trace, we may need the ability to talk about aggregations of traces. In 1981 Clarke introduced computation tree logic (CTL) [Clarke and Emerson, 1981], which has additional syntax and semantics for exactly that. The syntax of CTL is as follows:

- $A\phi$ - ϕ is true on all paths
- $E\phi$ - there exists a path on which p is true

We again define the semantics of CTL with respect to a Kripke structure $M = (S, S_0, R, L)$.

- $M, s \models \phi$ iff $\phi \in L(s)$.
- $M, s \models \neg\phi$ iff $\neg(M, s \models \phi)$.
- $M, s \models \phi_1 \wedge \phi_2$ iff $(M, s \models \phi_1) \wedge (M, s \models \phi_2)$.
- $M, s \models \phi_1 \vee \phi_2$ iff $(M, s \models \phi_1) \vee (M, s \models \phi_2)$.
- $M, s \models EX\phi$ iff for some state s', $R(s, s') \wedge M, s' \models \phi$.
- $M, s \models AX\phi$ iff for all states s', $R(s, s') \rightarrow M, s' \models \phi$.
- $M, s_0 \models A[\phi_1 U \phi_2]$ iff for all paths $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.
- $M, s_0 \models E[\phi_1 U \phi_2]$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.
- $(M, s \models AF\phi) \Leftrightarrow (M, s \models A[\top U \phi])$
- $(M, s \models EF\phi) \Leftrightarrow (M, s \models E[\top U \phi])$
- $(M, s \models AG\phi) \Leftrightarrow (M, s \models \neg EF(\neg\phi))$
- $(M, s \models EG\phi) \Leftrightarrow (M, s \models \neg AF(\neg\phi))$

In CTL, each A or E must be paired with an LTL operator. For example $AG\phi$, which says that ϕ must always hold on all paths. Alternatively, CTL* allows for free mixing of operators from LTL and CTL. This allows for terms such as $E(GF\phi)$, which is true iff there exists a path where ϕ will always be true at some future state. There is also ACTL*, which is CTL* with no existential branch quantifier.

2.2 Model Checking

Once the desirable properties of a system have be laid out in temporal logic, the next step is to verify that a program adheres to its specification. One approach is to manually (or with the assistance of a proof checker) construct a proof for the program. However, this costs the verifier in both time and mental effort and so a mechanised approach is desired. The first such automatic *model checker* was proposed by Clarke et al. [Clarke et al., 1986] to verify temporal properties of finite state programs. The algorithm they proposed was a search

based labelling of a finite state-transition graph, representing the program, with subformulas of the temporal logic specification.

Another approach is based on the notion that temporal logic properties can be expressed in terms of automata theory. Specifically, a finite state automaton over infinite words can be used to represent a temporal logic formula. Büchi automata [Büchi, 1962] are ω-automata, i.e. finite automata that accept an infinite stream of input, which may be constructed such that the automaton will accept exactly the inputs allowable by a temporal logic formula.

In [Vardi, 1996], the authors propose a model checking approach using this connection between temporal logic and automata theory. They propose the construction of a finite state, infinite word generator representing the program P, and an acceptor of the same, ϕ, constructed from the temporal property to be checked. Thus the program may be model checked by determining whether $P \cap \neg\phi$ is empty.

2.2.1 Büchi Automata

Like all ω-automata, the language of a Büchi automaton is ω-regular, i.e. a regular language extended to infinite streams. A regular language over the alphabet Σ is

- The empty language , *or*
- A singleton language $\{a\}$ for $a \in \Sigma$, *or*
- For two regular languages A and B:
  - $A \cup B$ the union of those languages, *or*
  - $A \,.\, B$ the concatenation of those languages, *or*
  - $A*$ the Kleene operation on that language.

The automata itself is defined as a tuple $A = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma \to Q$ is a transition function mapping states and letters to next states,

- $q_0 \in Q$ is an initial state, and
- $F \subseteq Q$ is a set of accepting states. A accepts an input stream iff it visits F infinitely often.

Rabin automata are also ω-automata, which are similar to Büchi automata except with the acceptance condition given by a set of pairs (E_i, F_i) such that an run is accepted if there is a pair where the run visits F_i infinitely often and does not visit E_i infinitely often.

2.3 Synthesis

Model checking is the art of deciding whether a program meets a specification. Synthesis is the related problem of building the program *to* a specification. In the model checking literature discussed thus far, the program and its environment are considered to be the same entity. This approach contains the assumption that the program and environment are cooperating, which is not appropriate for reactive synthesis. Consider a driver that must perform some action on input from the operating system. Driver synthesis with a cooperating OS could assume that the input never occurs. For synthesis of reactive programs an adversarial relationship more accurately formalises the problem. As such, the problem is reformulated as a two player game between the reactive program and its environment [Pnueli and Rosner, 1989].

A game is a structure $G = (S, \mathcal{U}, \mathcal{C}, \delta, s_0)$. We consider only two player games and we name those players the *controller* and *environment*. The structure is defined by:

- S is a finite set of states,
- $\mathcal{U}$ is a finite alphabet of *uncontrollable* actions,
- $\mathcal{C}$ is a finite alphabet of *controllable* actions,
- $\delta : S \times \mathcal{U} \times \mathcal{C} \to S$ is a transition relation mapping states, uncontrollable actions, and controllable actions to next states,
- $s_0 \in S$ is an initial state, and

Conceptually, the game structure is another finite state automaton where transitions are partially controlled by both players. In each state, the environment chooses an uncontrollable action from $\mathcal{U}$ and the system chooses a controllable action from $\mathcal{C}$. We consider only deterministic games where $\delta(s, u, c, s'_1) \wedge \delta(s, u, c, s'_2) \rightarrow (s'_1 = s'_2)$. We modify the notion of a run to suit games: $(s_0, u_0, c_0), (s_1, u_1, c_1), ...(s_n, u_n, c_n)$ where $\forall i[i \geq 0 \rightarrow \delta(s_i, u_i, c_i, s_{i+1})]$.

In addition to the game structure itself we define a game *objective* ψ given by an LTL formula. We say that a run is *winning* for the controller iff the run satisfies the objective. The game is zero-sum, therefore a run is winning for the environment in the dual case where the objective ψ is not true. Although ψ is given in LTL we formally define the objective of the game with the *all paths* branching operator: $\forall\mathcal{U}\exists\mathcal{C}A\psi$.

A *controller strategy* $\pi^c : \mathcal{S} \times \mathcal{U} \rightarrow \mathcal{C}$ is a mapping of states and uncontrollable inputs to controllable actions. π^c is a *winning strategy* iff all runs $(s_0, u_0, \pi^c(s_0, u_0)), (s_1, u_1, \pi^c(s_1, u_1)) \ldots$ are winning. *Realisability* is the problem of determining the existence of a winning controller strategy and *synthesis* is the problem of constructing it.

An *environment strategy* $\pi^e : \mathcal{S} \rightarrow \mathcal{U}$ is a mapping of states to uncontrollable actions. An environment strategy is winning iff all runs $(s_0, \pi^e(s_0), c_0)$, $(s_1, \pi^e(s_1), c_1) \ldots$ are winning for the environment. The existence of an winning environment strategy implies the nonexistence of a winning controller strategy and vice versa.

2.3.1 Solving Games

Reactive synthesis for a game with an LTL objective [Pnueli and Rosner, 1989] may be solved via the construction of an equivalent non-deterministic Büchi automaton that is subsequently determinised to a deterministic Rabin automaton. Without delving into details, the Rabin automaton is interpreted as a tree-automaton and checked for emptiness. This yields a double exponential time algorithm in the size of the specification.

The double exponential complexity, or *state explosion* problem, led to synthesis being considered infeasible for many years. However, synthesis has been applied in many real world scenarios by restricted the game objective to fragments of LTL. In this thesis we consider *safety games*, or games with objectives of the form $G\phi$ where ϕ is a propositional formula only. Informally, a controller in a safety game has the objective to stay within a safe region or

avoid error states. Safety games can be solved by drawing on μ-calculus and solving a fixed point calculation [Asarin et al., 1995], which is a much less complex procedure than the above automata theoretic approach.

2.3.2 Fixed Point Computation

Modal μ-calculus [Kozen, 1982] for propositional logic formalises the concept of fixed points. Given a monotone function f, a fixed point is a set X such that $f(X) = X$. The least fixpoint operator μ gives the smallest set X and the greatest fixpoint operator ν gives the largest. μ-calculus formulas have the following syntax, given with respect to a set of propositions P and a set of variables V.

- If $p \in P$ then p is a formula
- If p is a formula then $\neg p$ is a formula
- If p and q are formulas then $p \wedge q$ is a formula
- If p is a formula and Z is a variable then both $\nu Z.p$ and $\mu Z.p$ are formulas when all occurrences of Z have an even number of negations
- If p is a formula and Z is a variable then $\forall Z.p$ is a formula.
- Additionally, we introduce some syntactic equivalences:
  - $p \vee q \equiv \neg(\neg p \wedge \neg q)$
  - $\exists Z.p \equiv \neg \forall Z.\neg p$

Given a labelled transition system (S, F) where S is a set of states, and $F : P \to 2^S$ is a mapping of propositions to states with which they hold, we give the semantics of $\mu - calculus$ by a function $[\![p]\!]$:

- $[\![p]\!] = F(p)$
- $[\![\neg p]\!] = S \setminus [\![p]\!]$
- $[\![p \wedge q]\!] = [\![p]\!] \cup [\![q]\!]$
- $[\![\nu Z.p]\!] = \bigcup\{T \subseteq S | T \subseteq [\![p]\!][Z := T]\}$ where $[\![p]\!][Z := T]$ is $[\![p]\!]$ with Z mapped to T.

2.3.3 Fixed Point Game Solving

The safety game algorithm is centred on determining sets of states from which the controller can win. The building block of this algorithm is the uncontrollable predecessor ($UPre$), which returns a set of predecessor states from which the environment can force play into a given set. We define this operator for a game structure $G = (S, \mathcal{U}, \mathcal{C}, \delta, s_0)$ as

$$UPre(X) = \exists u \forall c \exists s \exists x [u \in \mathcal{U} \rightarrow (c \in \mathcal{C} \wedge s \in S \wedge x \in X \wedge \delta(s, u, c, x))]$$

For simplicity we describe the algorithm as though playing for the environment. As such we are actually solving the dual to the safety game: a reachability game. To solve the game we iteratively build a set of states backwards from the error set ($\neg\phi$) using the uncontrollable predecessor. It is clear that this set will grow monotonically and (since the state space is finite) eventually converge on fixed point. We call this fixed point set the environment's winning set. If this set contains the initial state then the game is unrealisable and the environment's winning strategy is to always play to stay within its winning set. Conversely, if the initial state is outside the environment's winning set then the controller must have a strategy to avoid the error states and the specification is realisable. To give the algorithm in terms of μ-calculus:

$$SAFE(T) = \nu Y. \neg UPre(Y \wedge \neg T)$$

2.3.4 Symbolic Algorithms

Model checking suffers from a similar state explosion issue and new approaches were developed in this field to deal with the problem. In the decisively titled *Symbolic model checking:* 10^{20} *states and beyond* [Burch et al., 1990] the authors claim an increase in the size of systems that can be checked from 10^3–10^6 to 10^{20}. This breakthrough result was achieved via symbolic, as opposed to explicit, representation of the states in the game structure. This result may also be applied to synthesis.

Consider the game structure in the previous sections. Without loss of generality, we may replace the set of states S with a set of boolean variables. A single state is now a valuation to those variables and a set of states may be symbolically represented by a boolean function. In order to combat state

explosion a compact representation of boolean functions is required. The standard choice is an ordered binary decision diagram (BDD) [Bryant, 1986].

BDDs represent boolean functions as directed acyclic graphs. Each node is contains a variable, each edge represents a decision (true or false) on its parent's variable. One node is designated root and each path through the graph will terminate in one of two sink nodes that represent whether the decisions on that path satisfy the boolean function or not. Isomorphic nodes (where both decisions lead to the same result) may be removed and isomorphic subgraphs may be merged in order to compress the function representation. The ordering of variables in the graph is important to this compressibility. In the worst case the representation is a tree with no removed or merged nodes, which is exponential in the number of variables. Given a boolean function and a variable ordering the corresponding BDD is canonical.

Conjunction and disjunction may be performed on BDDs via an algorithm with a running time of $O(n \times m)$ where n and m are the sizes of the two BDDs. The worst case of this algorithm is rarely reached however and in general the operation is efficient. Universal quantification may be performed by constructing the conjunction of the BDD with the quantified variable set one way and the BDD with the variable set the other way. Existential quantification works the same way with disjunction. In this way, the uncontrollable predecessor and winning sets may be efficiently computed and stored via BDDs.

2.3.5 Abstraction

When the state space is truly large, as it can be in many real world systems, symbolic representation is an insufficient optimisation to synthesise the system. Real world systems contain many complex details that may not be relevant to the verification property. Abstraction aims to reduce the level of detail in the system, without sacrificing correctness, so that it may be synthesised. For example, a system may require that a controller write a value to an 8 bit register (2^8 possible states). If the specification only refers to the register as being equal to a particular value then the abstraction may reduce this to 1 bit: set to the value, and not set to the value.

An abstraction is a mapping of *concrete* states onto a new, smaller, set of *abstract* states. Abstractions may be created manually or automatically constructed. A common technique is to approximate an abstraction for a system and refine it during the verification process. Counterexample guided

abstraction refinement (CEGAR) [Clarke et al., 2000] is a framework in which an approximate abstraction is refined via the analysis of counterexamples to the specification. An upper approximation is used for abstraction so that when the specification holds for the abstraction it also holds for concrete system. However, when a counterexample is found in the abstraction it may not be a concrete counterexample, in which case we call it a *spurious* counterexample. These counterexamples are used for refinement and the procedure begins anew with the refined abstraction.

2.4 Boolean Satisfiability

The ability to prove existentially quantified boolean formulas satisfiable or unsatisfiable (SAT) is enormously useful for program verification. Significant research has led to many highly efficient solvers for the SAT problem. Modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [Davis and Putnam, 1960; Davis et al., 1962]. This algorithm is a backtracking search that operates on the formula given in conjunctive normal form (CNF).

A CNF formula is a set of *clauses* of the form $(l_0 \vee l_1 \vee ... \vee l_n)$ where each *literal* l_i is a boolean variable or its negation. We call a clause with only one literal a *unit clause*. We call a variable *pure* if it appears in only one polarity in the formula. The DPLL algorithm propagates unit clauses and removes clauses with pure literals as its first step. Next a value is assigned to a variable and the algorithm recurses to search for a solution with that valuation. If none can be found then a backtracking occurs and the other polarity of the variable is tried. In modern solvers, clause learning is used to share information between different branches of the search tree. The algorithm terminates when the current variable assignment satisfies the formula, or the search space is exhausted.

2.4.1 Bounded Model Checking

In the previous section BDDs were discussed as a symbolic representation that compresses boolean functions and efficiently quantifies the function. A CNF representation of a function is not canonical and so does not necessarily suffer from the same state explosion problems as BDDs. Biere et al. [Biere et al., 1999] introduced a model checking methodology that takes advantage

of CNF as a symbolic representation and utilises a SAT solver to efficiently operate on it. We discussed BDDs in the context of solving games, however the context of this section is model checking of CTL* properties. It suffices to say that BDDs are applicable to this area as well and have similar advantages and disadvantages.

Instead of constructing a set of winning states this technique, called *bounded model checking*, searches for runs of the game. Conceptually the BDD approach is similar to breadth first search and bounded model checking is similar to depth first search. In broad strokes, the new methodology consists of constructing a propositional formula representing the existence of a program trace of a certain length k that violates the specification. The formula is solved by a SAT solver, which returns either SAT: a counterexample to the specification, or UNSAT: there is no counterexample of length k. The algorithm executes this process for increasing lengths k, which we call the *bound*.

One difficulty of this approach is choosing a maximum bound that is both *sufficient* to verify the program correct and *feasible* to compute the result of the propositional formula. For a finite state automaton, the *diameter* is the minimal length such that every reachable state can by reached by a path of that length or less. The diameter is sufficient for model checking but not always feasible. In addition, computing the diameter itself is an inefficient quantified boolean formula. Despite this difficulty, bounded model checking is useful in many practical cases. In particular the ability to quickly find short counterexamples gives an advantage in cases when a BDD based approach hits state explosion issues.

2.4.2 Interpolation

A Craig interpolant, $\mathcal{I}$, is defined with respect to two formulas, A and B, that are inconsistent ($A \wedge B \equiv \bot$) and has the following properties

- $A \rightarrow \mathcal{I}$
- $B \wedge \mathcal{I} \equiv \bot$
- $vars(\mathcal{I}) \subseteq vars(A) \cap vars(B)$ where $vars(X)$ is the set of variables referred to by X.

Propositional logic interpolants can be efficiently derived from the resolution proof of unsatisfiability of A and B. Due to their interesting properties and efficient construction, interpolants have be found to be useful in many areas of model checking [McMillan, 2005]. In general, interpolants are valuable for their ability to approximate. Intuitively, an interpolant is an approximation of A that captures only the details needed for a proof of unsatisfiability of A and B. If the proof represents something important about the system, such as a counterexample to the specification, then the interpolant captures important details. Interpolation can be used as an alternative to building a winning set in model checking by instead incrementally building an inductive invariant for the system from counterexample refutations. Chapter 3 contains a survey of model checking and synthesis methods that exploit this intuition. Interpolation is also central to the algorithm presented in this thesis.

3 Related Work

4 Bounded Realisability

In this chapter I will describe my work on bounded realisability of reactive systems with safety properties. As introduced in Chapter **??** reactive realisability is the problem of determining the existence of a program, which we call a *controller*, that continuously interacts with its environment in adherence with a specification. A safety property is a simple correctness condition that lays out a set of states of the system that controller must stay within. In this chapter we will refer to this property in the negation: the controller must avoid *error states*.

Realisability is the first step on the path to synthesis. In the subsequent chapter I will describe an algorithm that extracts the actions of the controller necessary for realisation. This strategy may be used for synthesis: automatic construction of the controller program. Reactive synthesis for controllers with safety properties has many practical uses in areas such as circuit design, device drivers, or industrial automation.

The algorithm described in this chapter solves bounded safety games. Recall that Chapter **??** introduced games as a formalism for synthesis. In this chapter we are concerned with *bounded* games that restrict all runs to certain length. This concept is borrowed from model checking where it is used for similar aims. Specifically, runs of a bounded game can be checked by a purely propositional formula passed to a SAT solver. The SAT solver provides an efficient method for quickly discovering counterexamples. The algorithm presented here is a counterexample guided abstraction refinement framework that relies on counterexamples to find and refine player strategies.

The primary motivation of this work is to avoid computing the entire set of winning states as would be done in the standard controllable predecessor driven fixpoint algorithm. An explicit representation of the winning set would

quickly run into the state explosion issues of synthesis so traditionally the set is represented symbolically with a binary decision diagram. However, a BDD is a canonical representation of a set that, in the worst case, may be exponential in the number of variables. For some systems there is no compact representation of the winning set and for those cases an algorithm that does not compute it, such as the one presented here, is desirable.

4.1 Inspiration

This work draws inspiration from a QBF solving algorithm that treats the QBF problem as a game [Janota et al., 2012]. In that algorithm one player assumes the role of the universal quantifiers and the opponent takes on the existential quantifiers. In the game, the players take turns to chooses values for their variables from the outermost quantifier block in. One method of solving QBF games is to expand the formula for each quantified variable by either conjuncting (for universal quantification) or disjuncting (for existential) the formula with the variable set to each possible value. For large QBF instances this expanded formula becomes far too large to solve so the authors introduce abstractions, or partially expanded formulas, to avoid expanding on variables unnecessarily. The abstractions are refined through a CEGAR process of searching for candidate solutions and analysing counterexamples. The full algorithm is described in detail in Chapter **??**. The bounded synthesis algorithm presented here takes inspiration from the CEGAR framework of this work and can be thought of as a domain specific version of the QBF algorithm.

4.2 Algorithm

Example. *We introduce a running example to assist the explanation. We consider a simple arbiter system in which the environment makes a request for a number of resources (1 or 2), and the controller may grant access to up to two resources. The total number of requests grows each round by the number of environment requests and shrinks by the number of resources granted by the controller in the previous round. The controller must ensure that the number of unhandled requests does not accumulate to more than 2. Figure 4.1 shows the variables (4.1a), the initial state of the system (4.1b), and the formulas for computing next-state variable assignments*

Controllable	Uncontrollable	State
request : {1, 2}	grant0 = {0, 1}	resource0 = {0, 1}
	grant1 : {0, 1}	resource1 = {0, 1}
		nrequests : {0, 1, 2, 3}

(a) Variables

```
resource0 = 0; resource1 = 0; nrequests = 0;
```

(b) Initial State

```
resource0' = grant0;
resource1' = grant1;
nrequests' = (nrequests + request >= resource0 + resource1)
           ?  (nrequests + request - resource0 - resource1) :  0;
```

(c) Transition Relation

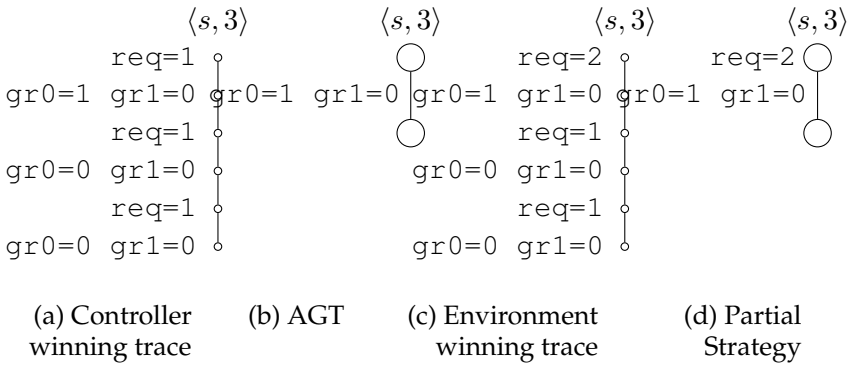
Figure 4.1: Example

(4.1c) for this example. We use primed identifiers to denote next-state variables and curly braces to define the domain of a variable.

This example is the $n = 2$ instance of the more general problem of an arbiter of n resources. For large values of n, the set of winning states has no compact representation, which makes the problem hard for BDD solvers. In Section 3 we will outline how the unbounded game can be solved without enumerating all winning states.

Our bounded synthesis algorithm constructs abstractions of the game that restrict actions available to one of the players. Specifically, we consider abstractions represented as trees of actions, referred to as *abstract game trees* (AGTs). Figure 4.2b shows an example abstract game tree restricting the environment (abstract game trees restricting the controller are similar). In the abstract game, the controller can freely choose actions whilst the environment is required to pick actions from the tree. After reaching a leaf, the environment continues playing unrestricted. The tree in Figure 4.2b restricts the first environment action to `request=1`. At the leaf of the tree the game continues unrestricted.

The root of the tree is annotated by the initial state s of the abstract game and the bound k on the number of rounds. We denote $\text{NODES}(T)$ the set of all nodes of a tree T, $\text{LEAVES}(T)$ the subset of leaf nodes. For edge e, $\text{ACTION}(e)$ is the action that labels the edge, and for node n, $\text{HEIGHT}(k, n)$ is the distance



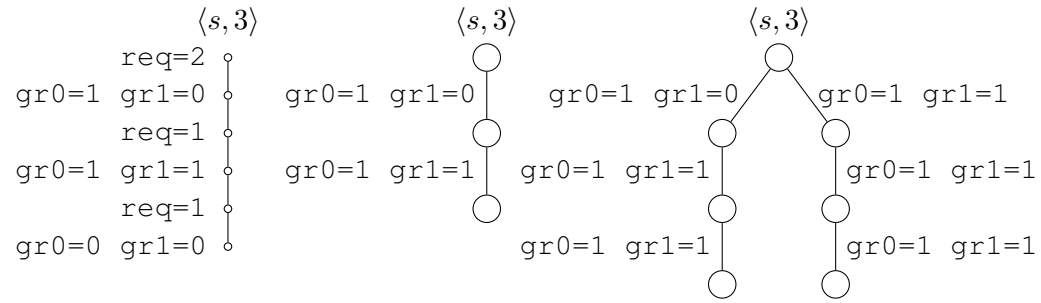
(a) Controller winning trace (b) AGT (c) Environment winning trace (d) Partial Strategy

Figure 4.2: Abstract game trees.

from n to the last round of a game bounded to k rounds. HEIGHT(k, T) is the height of the root node of the tree. For node n of the tree, SUCC(n) is the set of pairs $\langle e, n' \rangle$ where n' is a child node of n and e is the edge connecting n and n'.

Given an environment (controller) abstract game tree T a *partial strategy* $Strat$: NODES$(T) \rightarrow \mathcal{C}$ ($Strat$: NODES$(T) \rightarrow \mathcal{U}$) labels each node of the tree with the controller's (environment's) action to be played in that node. Given a partial strategy $Strat$, we can map each leaf l of the abstract game tree to $\langle s', i' \rangle$ = OUTCOME$(\langle s, i \rangle, Strat, l)$ obtained by playing all controllable and uncontrollable actions on the path from the root to the leaf. An environment (controller) partial strategy is *winning against* T if all its outcomes are states that are winning for the environment (controller) in the concrete game.

Example: Intuition behind the algorithm. *We present the intuition behind our bounded synthesis method by applying its* simplified version *to the running example. We begin by finding a trace of length k (here we consider $k = 3$) that is winning for the controller, i.e., that starts from the initial state and avoids the error set for three game rounds (see Figure 4.2a). We use a SAT solver to find such a trace, precisely as one would do in bounded model checking. Given this trace we make an initial conjecture that any trace starting with action* `gr0=1 gr1=0` *is winning for the controller. This conjecture is captured in the abstract game tree shown in Figure 4.2b. We validate this conjecture by searching for a counterexample trace that reaches an error state with the first controller action fixed to* `gr0=1 gr1=0`*. Such a trace, that refutes the conjecture, is shown in Figure 4.2c. In this trace, the environment wins by playing* `req=2` *in the first round. This move represents the environment's partial strategy against the abstract game tree in Figure 4.2b. This partial strategy is shown*



(a) Controller winning trace (b) First refined AGT (c) Final Refined AGT

Figure 4.3: Refined abstract game trees.

in Figure 4.2d.

Next we strengthen the abstract game tree taking this partial strategy into account. To this end we again use a SAT solver to find a trace where the contoller wins while the environment plays according to the partial strategy. In the resulting trace (Figure 4.3a), the controller plays `gr0=1 gr1=1` *in the second round. We refine the abstract game tree using this move as shown in Figure 4.3b. The environment's partial strategy was to make two requests in the first round, to which the controller responds by now granting an additional two resources in the second round.*

When the controller cannot refine the tree by extending existing branches, it backtracks and creates new branches. Eventually, we obtain the abstract game tree shown in Figure 4.3c for which there does not exist a winning partial strategy on behalf of the environment. We conclude that the bounded game is winning for the controller.

The full bounded synthesis algorithm is more complicated: upon finding a candidate partial strategy on behalf of player p against abstract game tree T, it first checks whether the strategy is winning against T. By only considering such strong candidates, we reduce the number of refinements needed to solve the game. To this end, the algorithm checks whether each outcome of the candidate strategy is a winning state for OPPONENT(p) by recursively invoking the synthesis algorithm on behalf of the opponent. Thus, our bounded synthesis algorithm can be seen as running two competing solvers, for the controller and for the environment.

The full procedure is illustrated in Algorithm 1. The algorithm takes a concrete game G with maximum bound κ as an implicit argument. In addition, it takes a player p (controller or environment), state s, bound k and an abstract

Algorithm 1 Bounded synthesis

function SOLVEABSTRACT(p, s, k, T)
  $cand \leftarrow$ FINDCANDIDATE(p, s, k, T) ▷ Look for a candidate
  if $k = 1$ **then return** $cand$ ▷ Reached the bound
  $T' \leftarrow T$
  loop
    if $cand =$ `NULL` **then return** `NULL` ▷ No candidate: return with no solution
    $\langle cex, l, u\rangle \leftarrow$ VERIFY($p, s, k, T, cand$) ▷ Verify candidate
    if $cex =$ `false` **then return** $cand$ ▷ No counterexample: return candidate
    $T' \leftarrow$ APPEND(T', l, u) ▷ Refine T' with counterexample
    $cand \leftarrow$ SOLVEABSTRACT(p, s, k, T') ▷ Solve refined game tree
  end loop
end function
function FINDCANDIDATE(p, s, k, T)
  $\hat{T} \leftarrow$ EXTEND(T) ▷ Extend the tree with unfixed actions
  $f \leftarrow$ **if** $p =$ `cont` **then** TREEFORMULA($k, \hat{T}$) **else** $\overline{\text{TREEFORMULA}}(k, \hat{T})$
  $sol \leftarrow \text{SAT}(s(X_{\hat{T}}) \wedge f)$
  if $sol =$ `unsat` **then**
    if `unbounded` **then** ▷ Active only in the unbounded solver
      if $p =$ `cont` **then** LEARN($s, \hat{T}$) **else** $\overline{\text{LEARN}}(s, \hat{T})$
    end if
    return `NULL` ▷ No candidate exists
  else
    return $\{\langle n, c\rangle | n \in \text{NODES}(T)\ , c = \text{SOL}(n)\}$ ▷ Fix candidate moves in T
  end if
end function
function VERIFY($p, s, k, T, cand$)
  for $l \in leaves(gt)$ **do**
    $\langle k', s'\rangle \leftarrow$ OUTCOME($s, k, cand, l$) ▷ Get bound and state at leaf
    $u \leftarrow$ SOLVEABSTRACT(OPPONENT(p), s', k', $\emptyset$) ▷ Solve for the opponent
    if $u \neq$ `NULL` **then return** $\langle$`true`$, l, u\rangle$ ▷ Return counterexample
  end for
  return $\langle$`false`$, \emptyset, \emptyset\rangle$
end function

game tree T and returns a winning partial strategy for p, if one exists. The initial invocation of the algorithm takes the initial state I, bound κ and an empty abstract game tree $\emptyset$. Initially the solver is playing on behalf of the environment since that player takes the first move in every game round. The empty game tree does not constrain opponent moves, hence solving such an abstraction is equivalent to solving the original concrete game.

The algorithm is organised as a counterexample-guided abstraction refinement (CEGAR) loop. The first step of the algorithm uses the FINDCANDIDATE function, described below, to come up with a candidate partial strategy that is winning when the opponent is restricted to T. If it fails to find a strategy, this means that no winning partial strategy exists against the opponent playing according to T. If, on the other hand, a candidate partial strategy is found, we need to verify if it is indeed winning for the abstract game T.

The VERIFY procedure searches for a *spoiling* counterexample strategy in each leaf of the candidate partial strategy by calling SOLVEABSTRACT for the opponent. The dual solver solves games on behalf of the opponent player.

If the dual solver can find no spoiling strategy at any of the leaves, then the candidate partial strategy is a winning one. Otherwise, VERIFY returns the move used by the opponent to defeat a leaf of the partial strategy, which is appended to the corresponding node in T in order to refine it in line (9).

We solve the refined game by recursively invoking SOLVEABSTRACT on it. If no partial winning strategy is found for the refined game then there is also no partial winning strategy for the original abstract game, and the algorithm returns a failure. Otherwise, the partial strategy for the refined game is *projected* on the original abstract game by removing the leaves introduced by refinements. The resulting partial strategy becomes a candidate strategy to be verified at the next iteration of the loop. In the worst case the loop terminates after all actions in the game are refined into the abstract game.

The CEGAR loop depends on the ability to guess candidate partial strategies in FINDCANDIDATE. For this purpose we use the heuristic that a partial strategy may be winning if each OUTCOME of the strategy can be extended to a run of the game that is winning for the current player. Clearly, if such a partial strategy does not exist then no winning partial strategy can exist for the abstract game tree. We can formulate this heuristic as a SAT query, which is constructed recursively by TREEFORMULA (for the controller) or $\overline{\text{TREEFORMULA}}$ (for the environment) in Algorithm 2.

The tree is first extended to the maximum bound with edges that are labeled with arbitrary opponent actions (Algorithm 1, line 14). For each node in the tree, new SAT variables are introduced corresponding to the state (X_T) and action (U_T or C_T) variables of that node. Additional variables for the opponent actions in the edges of T are introduced (U_e or C_e) and set to $\text{ACTION}(e)$. The state and action variables of node n are connected to successor nodes $\text{SUCC}(n)$ by an encoding of the transition relation and constrained to the winning condition of the player.

Algorithm 2 Tree formulas for Controller and Environment respectively

1: **function** TREEFORMULA(k, T)
2: **if** HEIGHT$(k, T) = 0$ **then**
3: **return** $\neg\mathrm{E}(X_T)$
4: **else**
5: **return** $\neg\mathrm{E}(X_T)\wedge$
6:

$$\bigwedge_{\langle e,n\rangle\in\text{SUCC}(T)} (\delta(X_T, U_e, C_T, X_n) \wedge U_e = \text{ACTION}(e) \wedge \text{TREEFORMULA}(k, n))$$

7: **end if**
8: **end function**
9: **function** $\overline{\text{TREEFORMULA}}(k, T)$
10: **if** HEIGHT$(k, T) = 0$ **then**
11: **return** $\mathrm{E}(X_T)$
12: **else**
13: **return** $\mathrm{E}(X_T)\vee$
14:

$$\bigvee_{\langle e,n\rangle\in\text{SUCC}(T)} (\delta(X_T, U_T, C_e, X_n) \wedge C_e = \text{ACTION}(e) \wedge \overline{\text{TREEFORMULA}}(k, n))$$

15: **end if**
16: **end function**

4.3 Optimisations

In this section we present several optimisations to the algorithm.

4.3.1 Bad State Learning

The most important optimisation that allows the algorithm to avoid much of the search space is to record states that are known to be losing for one player. On subsequent calls to the SAT solver we encode these states in the candidate strategy formula (see Algorithm 3). Thus the algorithm avoids choosing moves that lead to states that are already known to be losing.

Bad states are learned from failed attempts to find a candidate. If the SAT solver cannot find a candidate strategy for a given abstract game tree that means that there is a fixed prefix in the game tree for which the current player can never win. The state reached by playing the moves in the prefix must then be a losing state with some caveats. If the state is at the node with height k and losing for the environment then we know that the environment cannot force to the error set in k rounds. We do not know if the environment can force to the error set in $> k$ rounds. Therefore we record losing states for the environment in an array of sets of states B^e indexed by the height at which the set is losing. For the controller, a losing state is losing for any run of length $>= k$. In practical use we are uninterested in controller strategies that make use of states that would lose should the game be extended to a longer bound so we merely maintain a single set of controller losing states B^c.

Additional states can be learned by expanding a single state into a set of losing states by greedily testing each variable of the state for inclusion in a *cube* of states. This technique is well known in the literature and can be efficiently implemented using a SAT solver capable of solving under assumptions [Eén and Sörensson, 2003]. It is shown in Algorithm **??**.

4.3.2 Strategy Shortening

Learning new bad states means reducing the search space for the algorithm. It follows that it is better to learn states earlier in the algorithm's execution. One problem with relying on SAT calls that assume cooperation is that there is no urgency to the returned candidate strategies. Consider the running example: the environment can reach the error set by setting `request` to 2 during two rounds. However, in the empty abstract game tree of a bounded game of length 3 or longer, there is no reason for the SAT solver to make the first action one of the requesting rounds if it can assume the environment will never grant any resources. The first action is important because the candidate strategy

Algorithm 3 Modified Tree Formulas with Bad State Avoidance

function TREEFORMULA(k, T)
  if HEIGHT$(k, T) = 0$ **then**
    return $\neg B^c(X_T)$
  else
    return $\neg B^c(X_T) \wedge$

$$\bigwedge_{\langle e,n\rangle \in \text{SUCC}(T)} (\delta(X_T, U_e, C_T, X_n) \wedge U_e = \text{ACTION}(e) \wedge \text{TREEFORMULA}(k, n))$$

  end if
end function
function $\overline{\text{TREEFORMULA}}(k, T)$
  if HEIGHT$(k, T) = 0$ **then**
    return $\text{E}(X_T)$
  else
    return $B^e[\text{HEIGHT(K,T)}](X_T) \vee$

$$\bigvee_{\langle e,n\rangle \in \text{SUCC}(T)} (\delta(X_T, U_T, C_e, X_n) \wedge C_e = \text{ACTION}(e) \wedge \overline{\text{TREEFORMULA}}(k, n))$$

  end if
end function

is derived from that. The candidate is what the opponent has the chance to respond to, so if the candidate does not do anything useful the opponent's response has the freedom to be equally apathetic about reaching its goal. This leads to much of the search space being explored unnecessarily until we learn a losing state.

Encouraging the SAT solver to find *shorter* strategies is a successful heuristic for mitigating this issue. Whilst it does require more SAT calls per call to FINDCANDIDATE it can be efficiently implemented using incremental SAT solving and during our benchmarking we found the cost to be worthwhile. A strategy is shorter if following the strategy leads to a known bad state for the opponent is fewer game rounds. For the environment this is clearly analogous to reaching the error set sooner. For the controller it is less clear but intuitively states that are environment losing at a certain height are more likely to be *safe* states from which the controller may be able to force a loop.

Algorithm 4 Strategy Shortening

```
1:  function SHORTEN(p, s, k, T)
2:      T̂ ← EXTEND(T)
3:      f ← if p = cont then TREEFORMULA(k, T̂) else ¬TREEFORMULA(k, T̂)
4:      prev ← ⊤
5:      for l ∈ leaves(gt) do
6:          n ← ROOT(l)
7:          while HEIGHT(k, n) ≠ 0 do
8:              if p = cont then
9:                  a ← B^e[HEIGHT(k, l)](X_n)
10:             else
11:                 a ← E(X_n)
12:             end if
13:             sol ← SATWITHASSUMPTIONS(prev ∧ a, s(X_T̂) ∧ f)
14:             if sol ≠ NULL then
15:                 prev ← prev ∧ a
16:                 break
17:             end if
18:             n ← SUCC(n)
19:         end while
20:     end for
21:     return sol
22: end function
```

4.3.3 Default Actions

During the search for a candidate strategy the SAT solver selects actions for the opponent as though the players are cooperating. Sometimes the result is an action that will always fail for the opponent. In many specifications the environment is given the option to fail as a way of modelling errors. For example, in a network driver specification error transitions may be used to model failed connections. When such a transition exists it will often be selected by the SAT solver (especially when the strategy shortening optimisation is enabled). Constantly selecting a bad action for the opponent significantly affects the performance of the algorithm because no bad states can be learned and the solver must refine the game abstraction to avoid the bad action. Additionally, if a candidate strategy was found by relying on a bad action then it will usually need to be backtracked.

To avoid problematic action selection the solver can instead use some heuristic to select the arbitrary action required in the SAT call in FINDCANDI-

DATE. This does not affect the correctness of the algorithm. If no candidate can be found with the opponent playing an arbitrary action then clearly the selected action (or a different opponent action that is winning) would have eventually been refined into the abstract game if the opponent instead co-operated. A simple action selection heuristic has been observed to improve the performance of the solver during benchmarking. Before the main algorithm executes two SAT calls are made with formulas constructed from TREEFORMULA and $\overline{\text{TREEFORMULA}}$ called on an empty abstract game tree. From the result a mapping of height to *default action* is made for each player. During FINDCANDIDATE calls the arbitrary opponent actions are taken from the corresponding map at the appropriate height.

4.4 Correctness

Completeness of the algorithm follows from the completeness of the backtracking search. In the worst case the algorithm will construct the entire concrete game tree and effectively expand all quantifiers. Soundness follows from the existential search of the SAT solver in FINDCANDIDATE. The algorithm terminates after searching for a candidate strategy on an abstract game tree with moves fixed only for the opponent. If no candidate can be found with the opponent restricted in this way then no strategy exists for the player.

5 Synthesis with SAT

6 Evaluation

7 Conclusion

Bibliography

Eugene Asarin, Oded Maler, and Amir Pnueli. *Hybrid Systems II*, chapter Symbolic controller synthesis for discrete and timed systems, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. ISBN 978-3-540-47519-4.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proc. 1960 Internat. Congr .)*, pages 1–11. Stanford Univ. Press, Stanford, Calif., 1962.

J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, Jun 1990.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings*, chapter Counterexample-Guided Abstraction Refinement, pages 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-45047-4.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs*, pages 52–71, 1981.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. *Solving QBF with Counterexample Guided Refinement*, pages 114–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-31612-8.

Dexter Kozen. *Results on the propositional μ-calculus*, pages 348–359. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982. ISBN 978-3-540-39308-5.

Saul Kripke. Semantical considerations on modal logic. In *Acta Philosophica Fennica*, volume 16, pages 83–94. 1963.

K. L. McMillan. *Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings*, chapter Applications of Craig Interpolants in Model Checking, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31980-1.

Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 305–318, New York, NY, USA, 2011. ACM.

A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

Moshe Y. Vardi. *Logics for Concurrency: Structure versus Automata*, chapter An automata-theoretic approach to linear temporal logic, pages 238–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. ISBN 978-3-540-49675-5.