A Counterexample Guided Method for Reactive Synthesis

Alexander Legg

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Engineering

University of New South Wales

Sydney, Australia

August 2016

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES

Thesis/Dissertation Sheet

Surname or Family name: Legg		
First name: Alexander	**Other name/s: Jonathan**	
Abbreviation for degree as given in the University calendar:	**PhD**	
School: Computer Science and Engineering	**Faculty: Engineering**	
Title: A Counterexample Guided Method for Reactive Synthesis		

Abstract 350 words maximum: (PLEASE TYPE)

Software controllers of reactive systems are ubiquitous in situations where incorrectness has a high cost. In order to place trust in the software, strong guarantees of its functional correctness are required. Reactive synthesis can be used to automatically construct software to a specification and ensure correctness. The drawback is that synthesis is computationally hard and it is infeasible to synthesise a controller for many specifications.

Synthesis is formalised as a game between the controller and its environment. In this thesis we consider safety specifications that define the winning condition of the game for the controller as never allowing the game to visit an error state. The usual approach for solving controller synthesis is to compute the set of all winning states in the system and construct a controller that never deviates from this set. The set may be very large so it is standard practice to represent sets of states symbolically as a relation over the variables of the system. Binary decision diagrams (BDDs) are an efficient data structure used to store and manipulate sets of states for synthesis. The drawback of this approach is that a set of states has only one representation as a BDD and in some cases it may be exponentially large in the number of variables. The state explosion of BDDs causes controller synthesis to be infeasible on specifications with no compact representation of the set of winning states.

In this thesis I propose a synthesis algorithm that constructs an approximation of the set of safe states that is sufficient to show correctness of the controller. The algorithm constructs an abstraction of the game and searches for a candidate strategy for the controller. Counterexamples are used to refine the strategy until it is winning for the game abstraction. Similar to bounded model checking, a SAT solver is used to efficiently implement the search for a counterexample trace. When a strategy is found to be winning in the abstraction of the game an approximation of the states for which the strategy wins is extracted from the strategy via interpolation. The search continues by refining the abstraction until the approximation of winning states converges on a fixed point that is sufficient to prove that the specification is realisable.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral theses only).

.....	.....	.....
Signature	**Witness Signature**	**Date**

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY	**Date of completion of requirements for Award:**	

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed .....

Date .....

Abstract

Software controllers of reactive systems are ubiquitous in situations where incorrectness has a high cost. In order to place trust in the software, strong guarantees of its functional correctness are required. Reactive synthesis can be used to automatically construct software to a specification and ensure correctness. The drawback is that synthesis is computationally hard and it is infeasible to synthesise a controller for many specifications.

Synthesis is formalised as a game between the controller and its environment. In this thesis we consider safety specifications that define the winning condition of the game for the controller as never allowing the game to visit an error state. The usual approach for solving controller synthesis is to compute the set of all winning states in the system and construct a controller that never deviates from this set. The set may be very large so it is standard practice to represent sets of states symbolically as a relation over the variables of the system. Binary decision diagrams (BDDs) are an efficient data structure used to store and manipulate sets of states for synthesis. The drawback of this approach is that a set of states has only one representation as a BDD and in some cases it may be exponentially large in the number of variables. The state explosion of BDDs causes controller synthesis to be infeasible on specifications with no compact representation of the set of winning states.

In this thesis I propose a synthesis algorithm that constructs an approximation of the set of safe states that is sufficient to show correctness of the controller. The algorithm constructs an abstraction of the game and searches for a candidate strategy for the controller. Counterexamples are used to refine the strategy until it is winning for the game abstraction. Similar to bounded model checking, a SAT solver is used to efficiently implement the search for a counterexample trace. When a strategy is found to be winning in the abstraction of the game an approximation of the states for which the strategy wins is extracted from the strategy via interpolation. The search continues by refining the abstraction until the approximation of winning states converges on a fixed point that is sufficient to prove that the specification is realisable.

Acknowledgements

Acknowledge some people

Publications

- Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. A SAT-based counterexample guided method for unbounded synthesis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 364–382, 2016
- Niklas Eén, Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. SAT-based strategy extraction in reachability games. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3738–3745, 2015
- Nina Narodytska, Alexander Legg, Fahiem Bacchus, Leonid Ryzhyk, and Adam Walker. Solving games without controllable predecessor. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 533–540, 2014

Contents

1 Introduction

We rely on software systems to perform important tasks for us on a daily basis. Unfortunately we also frequently experience the frustration of an incorrect software system. However, as these systems become more ingrained into our lives the cost of incorrectness can be far greater than mere frustration.

In 1996 the European Space Agency lost their Ariane 5 rocket forty seconds after launch to an incorrect conversion from floating point to integer [Dowson, 1997]. The cost of the failure was $370 million in USD. More recently, Toyota has been forced to recall a large number of vehicles due to a failure in the software controlling the brakes [Parrish, 2013]. The failures led to loss of life [CBS News, 2010].

As the desire for software and the consequences of incorrectness has grown, the need for a systematic methodology for producing correct software has become apparent. One solution has been to develop strict engineering practices, including rigorous testing, to reduce the chance of errors. Another solution is to produce a proof of correctness of the software, either with or without the aid of a mechanised proof assistant. Model checking can also be used in some cases to automate the correctness proof.

A step further is to have our software automatically constructed for us, a technique first formally considered by Alonzo Church in the middle of the last century [Church, 1962]. Software synthesis shifts the role of the developer from writing code to writing formal specifications. This completely eradicates the human error factor from the low level construction of software and allows developers to focus on high level system design. In all other approaches to software correctness the software must first be constructed; a process involving considerable time and effort.

Unfortunately, automatic software synthesis involves nontrivial computa-

tion. In broad strokes, the synthesis algorithm must determine how the state of the system is affected by the software and its environment and then select actions for the software such that no matter the actions of the environment the system adheres to the specification. In practice, on certain system specifications the process can lead to significant *state explosion* that renders synthesis infeasible.

The state of the art in synthesis contains several methodologies that act as countermeasures to state explosion. However, no single approach is suited to all classes of specifications nor are all specifications currently feasible. In this thesis I propose a methodology for resisting state explosion on a set of synthesis specifications that are problematic for other approaches.

1.1 Synthesis

This thesis is concerned with synthesis of reactive systems. In a reactive system a controller interacts *continuously* with its environment by responding to inputs with the appropriate outputs. For example, a device driver is a reactive system in which the driver interacts with an operating system and a hardware device. Synthesising reactive systems like drivers is different to synthesising regular programs or functions since the correctness of a controller depends on how the system behaves over time instead of a single output corresponding to a single input. As a result, the reactive synthesis problem is staged as a game between the controller and its environment. For a detailed formalisation see Chapter 2.

This thesis is concerned with synthesis of controllers for safety games in which the winning condition for the controller is defined by ensuring that the game remains within a set of safe states. The game is zero sum, the environment wins if a state outside the safe set is reached. We say that we have *solved* a game if we can construct a winning strategy for one of the players. The usual approach to solving safety games is to iteratively construct a set of winning states that are known to be safe regardless of the actions of the environment. A winning strategy for the controller can be constructed by choosing actions that have successor states within the winning region.

Explicit enumeration of the states in the winning region is infeasible even on small specifications so the set of states is usually represented symbolically. This is done by specifying the game with states as valuations of a set of

boolean variables and using boolean algebra to symbolically define sets of states. Traditionally binary decision diagrams (BDDs) are used to represent boolean functions because they provide compact representations in most cases and there are efficient algorithms for operating on formulas in BDD form. The disadvantage of this approach is that in the worst case the representation occupies space that is exponential in the number of variables in the formula. A BDD is a canonical representation of a formula so it may be the case that a compact BDD representing the winning region for a particular specification does not exist.

Other approaches rely on satisfiability solvers to efficiently perform the operations required by synthesis on sets of states. The satisfiability problem (SAT) is the question of whether a value can be assigned to all variables in a formula such that the formula evaluates to true. Modern SAT solvers provide efficient implementations of backtracking search with computational learning that operate on boolean formulas in clausal normal form (CNF). The advantage of a SAT based approach is that CNF is not canonical so in cases when a BDD cannot compactly represent a set of states it may be possible to do so in CNF.

The disadvantage of SAT based approaches is that solvers only determine whether a satisfying assignment to variables *exists*. This is known as existential quantification. The dual problem, universal quantification, is to determine whether all variable assignments satisfy a formula. Both forms of quantification are required for synthesis in order to decide whether an action exists for one player that satisfies a property for all opponent actions. An example of this kind of computation would be deciding whether the controller can force the game into the winning region regardless of any action the environment chooses. It is possible to perform universal quantification with a SAT solver but it adds considerable complexity, which introduces another bottleneck to the synthesis process.

1.2 Approach

This thesis presents a SAT based approach that computes an approximation of the winning region. By approximating the winning region we hope to avoid the state explosion cost of representing the entire set of winning states. The algorithm is set within a counterexample guided abstraction refinement

framework. This is our approach to handling the alternating quantifications of synthesis. Candidate strategies are constructed and refined via counterexamples instead of precisely computing the result of the quantified formula.

In this approach, a SAT solver is used to verify whether a candidate strategy is a winning strategy for a safety game with a fixed number of a game rounds, which we call a bounded game. This approach is similar to bounded model checking where a program is verified by querying a SAT solver for a trace that violates the specification. In our bounded synthesis approach the SAT solver searches for a trace of opponent moves that cause the candidate strategy to lose the bounded game. As with bounded model checking, a counterexample trace informs the algorithm how to refine the candidate strategy.

Discovering a winning strategy for the bounded game does not guarantee that the strategy is winning in the unbounded game. Specifically, if the controller strategy avoids an error state for k rounds this does not guarantee that it can avoid errors for $k + 1$ rounds. We address this problem with an extension to the algorithm that iteratively solves bounded games while incrementing the bound. During the execution of the bounded game solver we learn losing states for both players. This computational learning serves a dual purpose by both serving as an optimisation to reduce the search space and also providing the termination condition. By carefully learning states that are losing for the environment we may construct an overapproximation of the environment's winning region. The overapproximation can be used to guarantee that the actual winning region does not contain the initial states and so there cannot be a winning strategy for the environment.

1.3 Contribution

This thesis presents a SAT based counterexample guided approach to controller synthesis of safety specifications. This approach includes a bounded synthesis algorithm, an extension to unbounded synthesis, and a methodology for extracting strategies from the certificate generated by the bounded synthesis algorithm.

The approach is designed to solve synthesis specifications where the winning region is difficult to represent compactly with existing symbolic techniques. The aim of this work was not to produce a one size fits all approach

to safety synthesis but instead to provide a solution suited to some of the problem instances that are difficult to solve for other methods.

The instances that emit winning regions that are difficult to efficiently represent with binary decision diagrams include many real world problems. An example of such a specification is an arbiter that must grant resources from a homogeneous pool in order to fulfil requests from the environment. In this problem the winning region for the environment must exclude all combinations of resource allocations that exceed the number of requests. There is no compact representation of this kind of winning region as a binary decision diagram but in our approach we use overapproximation so that we don't need to find and represent all combinations.

In order to validate the methodology I have implemented the algorithm as an open source tool. In later chapters we present benchmarks that show that the algorithm is promising and although it does not solve as many problem instances as other techniques it performs better on certain classes of problems.

1.4 Summary

- *Reactive synthesis* can be used to automatically generate correct-by-construction controllers for software systems. Compared to other approaches to software correctness synthesis does not require the software to first be developed.

- Synthesis is formalised as a game between a controller and its environment. In many cases these games can be solved by constructing a symbolic representation of the winning states of the game using a binary decision diagram. However, for some games there is no compact representation of the winning region.

- This thesis presents a SAT based counterexample guided approach that targets these cases by constructing an approximation of the winning region that is sufficient to determine the winner of the game.

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

In this thesis we are concerned with reactive systems. Traditional programs can be verified by checking that the output is correct for each possible input. In a reactive system, i.e. a system that is in a continuous state of interaction with its environment, there is no termination and therefore no final output to verify. Instead the system is 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.

In this thesis, device drivers will be frequently used as an example of a reactive system. A driver accepts requests from the operating system and information about internal state from the device, and it responds by sending commands to the device and reporting to the operating system. The correctness of a device driver might be specified by a statement in temporal logic that corresponds to something similar to *the driver does not enter an error state* or *the driver always responds to requests*. In the following sections the syntax and semantics required to make such statements formal will be introduced.

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 that must hold in particular states. What is now lacking is a means of bringing states together to express properties of 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 we 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

Before turning our attention to the synthesis of a program that is correct according to its temporal logic specification let us consider the simpler problem of verifying that an existing program is correct. Verification can be done by manually constructing a proof of correctness but this is labour intensive process even with the assistance of a mechanised proof assistant. Here we consider *model checking*, which is the problem of automatically verifying the system.

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 checked by determining whether $P \cap \neg\phi$ is empty.

In Section 2.4.1 another approach to model checking is discussed that searches for counterexamples to safety properties using a SAT solver.

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 automaton 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 *constructing* the program to meet a specification. In model checking the actions of the program are decided by the software being checked. A synthesis procedure must instead decide how the controller chooses actions and the ways the environment can react to each decision. In order to model this requirement the controller and environment should be considered to be in an adversarial relationship. Thus the synthesis problem is formulated 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.

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 $\delta(s, u, c, s'_1) \wedge \delta(s, u, c, s'_2) \to (s'_1 = s'_2)$. 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 $\forall i[i \geq 0 \to \delta(s_i, u_i, c_i, s_{i+1})]$.

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. For a controller to be correct it must ensure that all choices of the environment lead to runs that are winning for the controller.

A *controller strategy* $\pi^c : \mathcal{S} \times \mathcal{U} \to \mathcal{C}$ 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* $\pi^e : \mathcal{S} \to \mathcal{U}$ 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 a 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 causes a *state explosion*, which led to synthesis being considered infeasible for many years. However, synthesis has been applied in many real world scenarios by restricting 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.

Whilst the LTL fragment that is solvable via a safety game is not very expressive, fixed point computations can also be used to solve the much more expressive generalised reactivity fragment of LTL. Safety synthesis can be seen as the first step on the path to more useful yet practical synthesis techniques.

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

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

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. Figure 2.1 shows two example BDDs that represent the same boolean function with different variable orderings.

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.

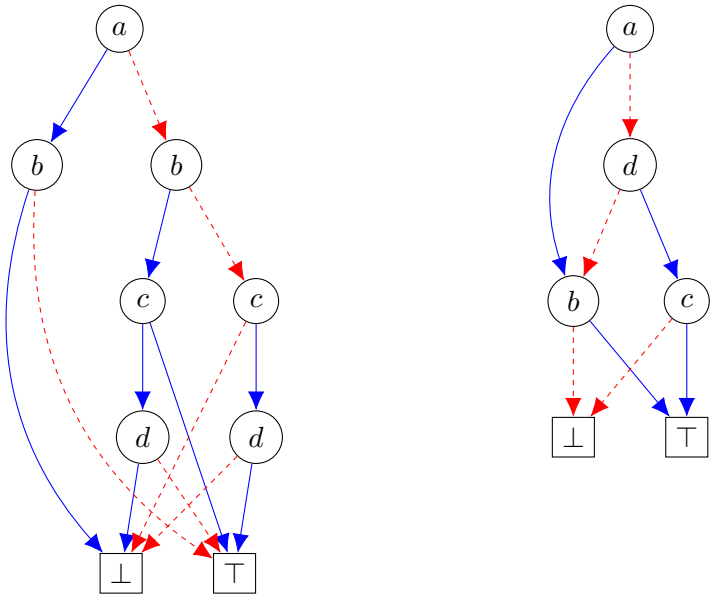


Figure 2.1: BDDs for $(\neg a \vee b) \wedge (a \vee c \vee \neg d) \wedge (b \vee d)$.
Solid blue transitions are 1, dashed red transitions are 0.

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.

2.4.3 Quantified Boolean Formulas

A quantified boolean formula (QBF) extends satisfiability with the addition of quantifiers. We consider formulas in prenex normal form $Q_1\hat{x}Q_2\hat{y}...F(\hat{x},\hat{y},...)$ where $Q_i \in \{\exists,\forall\}$, $\hat{x},\hat{y},...$ are sets of boolean variables, and F is a propositional formula over the quantified variables in CNF.

Quantifiers over boolean variables may be *expanded* into propositional formulas like so:

- $\exists x F(x) \equiv F(\texttt{true}) \vee F(\texttt{false})$
- $\forall x F(x) \equiv F(\texttt{true}) \wedge F(\texttt{false})$

Expansion may be used to construct a SAT problem from QBF but the CNF formula may be exponentially larger than its QBF representation. We will discuss alternative approaches to solving QBF problems in Chapter 3.

A *Skolem function* for variables $\hat{s}$ with respect to a QBF

$$\exists\hat{x}_1\forall\hat{y}_1...\exists\hat{x}_i\forall\hat{y}_i\exists\hat{s}Q_1\hat{z}_1...Q_j\hat{z}_jF(\hat{x}_1,\hat{y}_1,...,\hat{x}_i,\hat{y}_i,\hat{s},\hat{z}_1,...,\hat{z}_j)$$

is a function $f : 2^{|\hat{y}_1|} \times ... \times 2^{|\hat{y}_i|} \rightarrow 2^{|\hat{s}|}$ such that

$$\exists\hat{x}_1\forall\hat{y}_1...\exists\hat{x}_i\forall\hat{y}_iQ_1\hat{z}_1...Q_j\hat{z}_jF(\hat{x}_1,\hat{y}_1,...,\hat{x}_i,\hat{y}_i,f(\hat{y}_1,...,\hat{y}_i),...,\hat{z}_1,...,\hat{z}_j)$$

is equisatisfiable to the original QBF. In other words, if ψ is a satisfiable QBF, then f assigns a value to $\hat{s}$ for every assignment to universally quantified variables in the prefix such $\psi[\hat{s}/f]$ is also satisfiable. It is possible to reduce a QBF to *Skolem normal form* by substituting all existential variables with a valid Skolem function.

2.5 Summary

In this chapter we have introduced several concepts necessary for an understanding of the central work of this thesis. We briefly summarise some key points here as an aid to the reader.

- Temporal logic is the language we use to describe systems when a formalisation of time is necessary to express correctness. The existing body of work on synthesis and model checking is vast and this mathematical foundation of temporal logic provides the common language we use to define particular specialisations. In this thesis we are concerned with safety synthesis, which is formalised as a two player game with a winning condition defined by a subset of linear temporal logic with a single global operator.

- Model checking is an approach to automatically verifying the correctness of programs. In this chapter we briefly introduced the problem and two techniques used to solve it. In the next chapter we will expand on some existing work on model checking that is related to the synthesis techniques introduced by this thesis.

- Synthesis is a game between a system and its environment. Synthesis games may be solved by constructing a winning region via a fixed point computation of the controllable predecessor operator. In order to make this process scalable states are represented symbolically. Binary decision diagrams, which are graphical representations of boolean formulas, are commonly used as a compact symbolic representation of a set of states.

- The algorithm presented in this thesis constructs boolean formulas. Here we have defined the problem of satisfiability for boolean formulas and introduced the tools that solve them. Additionally the algorithm makes use of interpolation of boolean formulas, which can be performed efficiently using the certificate generated by SAT solvers.

3 Related Work

Reactive synthesis is an extensively studied topic and the work of this thesis is influenced by a wide array of prior work. In the previous chapter we identified symbolic representation of state sets and abstraction refinement as methodologies for mitigating state explosion. In this chapter we will approach the problem from a different angle. The work in this thesis is inspired by research in the model checking community and some prior efforts to apply that research to synthesis.

3.1 Bounded model checking

Bounded model checking [Biere et al., 1999] (BMC), as introduced in Chapter 2.4.1, is a methodology that generates SAT queries to determine the existence of a trace in a game that violates its specification. The approach taken to realisability described in Chapter 4 is inspired by BMC and also unrolls the transition relationship into a SAT query and searches for counterexample traces.

BMC considers the validity of CTL* formulas in Kripke structures in which the game is bounded to k rounds. The authors of the procedure provide a semantics for the translation of CTL* formulas on bounded models, via LTL, to satisfiability constraints. Consider a safety property $AG\phi$ as an example. A safety property is universal and so is checked in BMC by searching existentially for counterexamples in the form of the negation $F\neg\phi$. The search is translated into a SAT query by unrolling the transition relation R like so: $s_0 \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$. The LTL formula is similarly translated into a formula $\bigvee_{i=0}^{k} \neg\phi \in L(s_i)$. The SAT query is equivalent to a check whether there is

a path in the bounded Kripke structure $(s_0, s_1, ..., s_k)$ such that $\neg\phi$ holds in some state s_i.

For bounded realisability, the unrolling of the transition relation and the translation of the safety property can be done in much the same way. The difference is that in a synthesis problem the model has yet to be constructed and the algorithm must be given the freedom to choose the actions of the controller. For more details about how this can be overcome, see Chapter 4.

One of the motivations behind bounded model checking is aligned with the aim of this thesis: to avoid the high cost in space of approaches that construct a symbolic representation of the winning region as a binary decision diagram. By bounding the length the game the procedure can rely on SAT as a symbolic representation instead of BDDs. The drawback is that although BMC is sound, any counterexample is a true counterexample, it is not complete with respect to the unbounded game unless a sufficient bound is used. For safety properties the diameter of the game gives a tight sufficient upper bound for BMC although it is difficult to compute. Due to the additional complexity of synthesis it is not feasible to use the diameter in a bounded synthesis procedure.

3.2 Unbounded model checking

The usage of SAT solvers in bounded model checking proved to be highly beneficial for discovering counterexamples. Research into applications of SAT in unbounded model checking has subsequently progressed in several directions.

3.2.1 Non-canonical symbolic representation

One approach to unbounded model checking is to replace BDDs with binary expression diagrams (BEDs) or reduced boolean circuits (RBCs) in a fixed point algorithm [Abdulla et al., 2000; Williams et al., 2000]. BEDs are a generalisation of BDDS with the advantage that BEDs are not canonical and their use as a symbolic representation may be more succinct than the equivalent BDD. An RBC is simply a graphical representation of a circuit with some reductions applied. The two representations are essentially orthogonal and conversion between them is linear.

The disadvantage is that the controllable predecessor computations during the fixed point calculation require quantifier elimination that increase the size of the BED or RBC. Detecting a fixed point in the state sets then requires a costly satisfiability check of combinations of the expanded formulas. One option is to construct an equivalent BDD for which the satisfiability check is efficient but potentially negates the advantage of the non-canonical representation. It is also possible to construct a CNF representation of the formula from either a BED or RBC and query a SAT solver for satisfiability. Neither option fully mitigates the potential size of the expanded formula due to quantifier elimination. In practice this methodology works well only on models with few inputs so that quantification does not explode the formulas.

3.2.2 Hybrid SAT/BDD approach

Although many research efforts are directed away from BDDs to avoid their space blowup some work instead focuses on allowing a trade off between space and time via combinations of SAT procedures with BDDs. One such approach [Gupta et al., 2000] uses BDDs to enhance SAT in two ways during a reachability fixed point algorithm.In Chapter 2 a fixed point algorithm for solving safety games using the uncontrollable predecessor backwards from the error set was introduced. The authors of this work solve the dual game, reachability, in a forwards direction by iteratively computing the *image* of the initial states. The image operator takes a set of source states and returns the states that one player can force the game to. This is the forward searching version of the uncontrollable predecessor, which is also sometimes called a *preimage*.

The authors introduce a technique they call *BDD bounding* to prune the search space of the SAT procedure by checking that partial assignments to a set of variables during the SAT search are contained within a BDD. An image computation requires that the assignment to current state variables is contained within the source states computed during the previous iteration. The set of source states may be represented as a BDD and BDD bounding applied to the search in order to detect and immediately backtrack when a satisfying assignment has current state variables set to a value outside the BDD.

It is possible to directly apply SAT to quantifier elimination, and thus to image computation, by repeatedly applying a SAT solver to find all satisfying

solutions. This methodology applied without any optimisations is generally infeasible due to the large number of calls that must be made to the SAT solver. The authors of [Gupta et al., 2000] introduce a middle ground between this entirely SAT based approach and a standard BDD image computation. They suggest interrupting the SAT procedure after some partial assignment has been made to continue computation with a BDD. Effectively this is BDD image computation but distributed into smaller components by the SAT solver.

3.2.3 SAT based unbounded model checking

An optimised approach to SAT image computation is an efficient model checking procedure is cases where cube enumeration does not cause exponential blowup [McMillan, 2002]. McMillan proposes constructing *blocking clauses* by modifying the SAT procedure and analysing the solver's internal implication graph. The result is effectively cube enumeration that produces a CNF formula with intelligently enlarged cubes so that the original formula is covered in fewer SAT calls. The procedure is applied to CTL model checking by performing universal quantification on CNF formulas via variable deletion.

3.2.4 Application of Craig interpolants

Another angle of research is to extend bounded model checking into an unbounded procedure via a more efficient means than computing a diameter or other sufficient bound. In [McMillan, 2003] Craig interpolation is proposed as means of approximating the set of reachable states during bounded model checking.

Recall the introduction of Craig interpolants in Section 2.4.2. An interpolant is a formula that may be constructed efficiently from the resolution proof of two mutually unsatisfiable formulas. It is implied by one formula and the conjunction with the second formula is unsatisfiable. During bounded model checking a formula representing an unrolling of the transition relation of length k is unsatisfiable if there is no counterexample trace. This formula may be separated into an initial game round and $k-1$ remaining game rounds enabling a convenient application of interpolation. An interpolant constructed this way is an overapproximation of the image computation on the initial states and the states contained in the interpolant cannot emit a counterexample in $k-1$ game rounds.

This process is applied iteratively by setting the starting point of the unrolled formula to the union of the initial states and any previously computed interpolants. By construction via interpolants no state inside this set enables a counterexample run. Eventually this set will either reach a fixed point and provide an inductive invariant of the system, indicating that the specification of the model holds in the unbounded game, or a counterexample run will be found.

3.2.5 Properly Directed Reachability (PDR)

More recently an approach was proposed that solves safety games without unrolling the transition relation of the game [Bradley, 2011]. The intuition of the algorithm is to construct a proof of a safety properly P by incrementally strengthening a series of inductive lemmas. This procedure provided the inspiration for the unbounded realisability algorithm of Chapter 6.

The algorithm maintains a series of formulas $F_0, F_1, ..., F_k$ that overapproximate the set of states reachable in $0, 1, ..., k$ game rounds. The sequence is extended when $F_k \wedge T \rightarrow P'$ is true indicating that $F_{k+1} = P$ is a new reachable set that maintains the safety property. Then clauses in each F_i are propagated forwards to F_{i+1} if it is possible to do so. When P is not reachable from F_k there must be a state within F_k that is one step from violating the safety property. Either this state indicates a counterexample or a new relatively inductive clause can be added to some F_i to prevent the state from being reachable at F_k. If during the forward propagation of clauses two formulas F_i and F_{i+1} become equivalent the algorithm has reached a fixed point and has proved that the safety property is invariant.

3.3 Synthesis with SAT

Given the success of model checking techniques employing satisfiability methods it is no surprise that many attempts have been made to replicate these results in the context of synthesis. Synthesis is a significantly more complex problem and it is not obvious how to translate the advantages of SAT, namely the ability to quickly find counterexamples, to an algorithm that must construct a model before checking it. Nonetheless advances have been made that are able to outperform BDD methods in some situations.

3.3.1 Bounded Synthesis

In Chapter 4 we will discuss a bounded realisability algorithm. The bounded synthesis methodology introduced by [Finkbeiner and Schewe, 2013] is an unfortunate conflation of terms. Their approach places a bound on the size of the implementation as opposed to bounding the length of the game as in Chapter 4 and in bounded model checking.

This bounded synthesis approach is used to synthesise reactive systems for distributed architectures by first constructing a universal co-Büchi automaton for the given LTL specification. An implementation is a transition system that drives that automaton thereby producing a run graph. The run graph of a transition system may be annotated in each node with the maximal number of rejecting states that occur on any path to that node. The authors show that the existence of an annotation with finite bounds indicates that its transition system is accepted by the automaton and hence the LTL specification. A bound is placed on the size of the transition system, which additionally sets an upper bound for the maximum label in the annotation, and an SMT solver can be used to search for a bounded transition system that has a valid annotation. In this way LTL synthesis is reduced to a series of SAT modulo integer arithmetic problems with increasing bounds.

One synthesis tool [Ehlers, 2012] divides an LTL specification into safety and non-safety components. The safety components are solved easily by a standard symbolic algorithm with BDDs. The author proposes a symbolic version of bounded synthesis to solve the non-safety components. Their approach constructs a BDD that encodes the search for a transition system with a valid annotation. Another approach [Filiot et al., 2011] similarly does symbolic bounded synthesis using antichains.

3.3.2 Lazy Synthesis

A counterexample guided framework has been applied to bounded synthesis in a methodology called lazy synthesis [Finkbeiner and Jacobs, 2012]. The authors propose the construction of bounded size partial implementations via SMT solving a collection of constraints. The partial implementation is then model checked in a symbolic BDD algorithm and any counterexamples are used to introduce new constraints that refine the partial strategy. If the implementation is found to be correct during the model checking phase then

the algorithm terminates. Alternatively, the constraint solver may return that there is no implementation at which point the bound on the size of the implementation is increased.

The counterexample guided search for a correct implementation is similar to the bounded realisability approach proposed in Chapter 4. The framework is fundamentally the same: candidate strategies are found by a SAT solver, they are checked for correctness, and counterexamples are used to refine further searches for candidates. However, the two methodologies use different approaches to each component of that framework.

3.3.3 Properly directed reachability applied to synthesis

The incremental induction of PDR [Bradley, 2011] (see Section 3.2.5) has also been applied to the realisability problem. In [Morgenstern et al., 2013] the authors suggest that by replacing the SAT queries used to approximate reachability with 2QBF queries that the algorithm may be used to solve realisability of safety games.

Their approach computes overapproximations of the sets of states from which the environment can force to an error state in some number of game rounds. A state is added to the overappoximation of states that are environment winning in k rounds via a 2QBF query that checks whether the environment has an action such that for all controller actions a successor state is inside the overapproximation of states winning in $k-1$ rounds. This generates new obligations for the algorithm to refine the overapproximations. Each successor state must now be checked for the ability for the environment to win in $k-1$ rounds. Eventually this process may discover a chain of states from the initial set to the error set such that the environment can force a win. Alternatively the overapproximating sets will reach a fixed point indicating that the controller can force the game to stay within a set of safe states.

The universal quantification in the 2QBF query is costly to compute so the authors propose repurposing SAT for the task. Similar to how QBFs are solved in [Janota et al., 2012] and in Chapter 4, a SAT query checks whether there is an existentially quantified pair of controller and environment actions that reaches the desired set, and another SAT query gives the controller the opportunity to revise its action. Intuitively, the first query *guesses* an environment transition and the second query *checks* it. To assist the process an overapproximation of controller winning states is maintained and used to direct the controller away

from its losing states in the checking query. Additionally, the environment transitions that turn out to be bad guesses are learned and blocked in future attempts.

A recent approach [Chiang and Jiang, 2015] has a similar application of PDR to synthesis with the major difference being that the authors propose solving the game forwards from the initial states instead of backwards from the error set. Thus the relatively inductive sets represent overapproximations of reachable states. In the previous work the SAT query checks for environment actions that force a successor state into an approximation of environment winning states. As a result, when a transition is found to have a countering controller action it is only known to be a bad transition for the environment to force into the current target. In this more recent work the SAT query is always attempting to find transitions *from* the (approximate) reachable sets into the error set. An advantage of this approach is that learned transitions may be blocked in *all* future queries.

3.3.4 Clause Learning for Synthesis

In [Bloem et al., 2014] the authors propose a suite of learning algorithms for synthesis. Two of these algorithms are centred on learning unsafe states by solving a quantified formula that checks for environment controllable successor states outside the current approximation of the safe region. When such a state is discovered it is generalised into multiple cubes representing sets of states that are then blocked from the safe region. The specification is decided unrealisable when the initial states are no longer within the safe region and realisable when there are no states left to learn.

The two variations of this algorithm correspond to one based on a QBF solver and one based on two competing incremental SAT solvers. The latter contains an optimisation to ensure that the incremental nature of the solvers is exploited. Incremental solvers work well in the case where new constraints are added to the problem over time. Removing constraints requires either careful backtracking of learned clauses or restarting the session with no clauses. As the safe region is restricted by blocking cubes the constraints of the solver playing on the behalf of the environment are reduced by enabling the search for transitions to outside the safe region to visit the blocked cubes. The authors suggest maintaining a separate instance of the safe region that is lazily updated. The environment searches for states with successor states outside

the old version of the safe region until it is necessary to make the costly update to the incremental solver to the more permissive new safe region.

Another optimisation take inspiration from PDR to approximate reachability information during clause learning. It is not useful to learn unreachable states to remove from the safe region so the search space can be pruned by only considering an overapproximation of reachable states. The optimisation is implemented first by checking candidates for learning for inclusion in the initial set or a predecessor inside the current safe region estimate.

The authors additionally propose two approaches that attempt to directly compute a winning region. The first of these searches for assignments to the parameters of a CNF template of the winning region with a call to a QBF solver. The parameters correspond to the polarity and inclusion of variables within clauses. The second constructs an *effectively propositional logic* (EPR) formula that characterises the Skolem functions encoding the winning region of the game. This cannot be solved via QBF due to the nonlinear nature of quantifiers over current and successor variables that describe the winning region. The formula can be encoded in EPR and handed to an efficient solver.

Each of these algorithms has been implemented in a tool that has the ability to run various combinations in parallel. The authors report a significant benefit to parallelisation and sharing of learned clauses in between algorithms.

3.4 Quantified Boolean Formula Solving

A quantified boolean formula (QBF) generalises the satisfiability problem to include universal and existential quantifiers. In accordance with the additional complexity of quantification QBF solvers have so far been less successful than SAT solvers at scaling to real world problems. However recent work in which competing SAT solvers are employed on behalf of each quantifier in a QBF problem have shown promising advances.

The bounded realisability problems of Chapter 4 are specialisations of QBF problems. The algorithm proposed to solve those problems can be seen as a domain specific QBF solver. In particular, the algorithm is similar to and inspired by the general QBF algorithm of [Janota et al., 2012]. In this section we will review the state of the art in QBF solving in order to shed light on the advantages that an algorithm such as in Chapter 4 has over a generalised QBF solver.

QBFs may be solved by application of DPLL [Cadoli et al., 1998] or by *expansion* into SAT [Ayari and Basin, 2002] (see Section 2.4.3). In the former, which we refer to as *search-based* approaches, computational learning can be applied to share information between separate branches of the search tree [Giunchiglia et al., 2002; Zhang and Malik, 2002]. In SAT conflicts are be remembered in order to prune the search space but once a satisfying assignment is found the solver may terminate. A QBF solver must explore satisfiability in multiple branches of the search in order to establish truth for every value to universal variables. This additional aspect to search may also be reduced by retaining *cubes* of assignments that are known to lead to satisfiability.

3.4.1 Q-resolution

Q-resolution [Büning et al., 1995] is a method for combining clauses and eliminating variables to eventually solve a QBF. We consider QBFs in prenex normal form with quantifiers $Q_1\hat{x}_1Q_2\hat{x}_2...Q_n\hat{x}_n$. We assign an ordering to variables corresponding to its scope: $\hat{x}_1 < \hat{x}_2 < ... < \hat{x}_n$. We say that a formula is *forall reduced* if each universally quantified literal l is deleted from clauses with no existentially quantified literals of larger scope. This reduction preserves equivalence and ensures that the innermost quantifier is always existential. For example, $\forall x_1 \exists y_1 \forall x_2((x_1 \vee y_1 \vee x_2) \wedge (x_1 \vee \neg y_1 \vee \neg x_2))$ is equivalent to $\forall x_1 \exists y_1((x_1 \vee y_1) \wedge (x_1 \vee \neg y_1))$.

Q-resolution is used to generate new clauses for a forall reduced QBF. Two existing clauses are selected with opposite polarities of an existentially quantified variable y. Taking the union of literals of both clauses, removing y literals, and reapplying forall reduction produces a new clause called the *resolvent*. If all possible resolvent clauses are generated for a variable y it may be removed entirely from the formula along with any clause containing y. For example, $\forall x_1 \exists y_1((x_1 \vee y_1) \wedge (x_1 \vee \neg y_1))$ may be further reduced to $\forall x_1(x_1)$ by resolving on y_1, which after forall reduction is the empty clause and the QBF is shown to be false.

In practice, solving a QBF with q-resolution alone will generate too many clauses to be feasible. In [Biere, 2005] an approach combining q-resolution with expansion is suggested. Resolution is used to eliminate variables from the innermost existential scope and expansion for the innermost universal scope. A scheduler selects which variable to eliminate next by selecting from all

candidate variables the variable with the lowest cost. The cost of eliminating a variable is set to the upper bound of the number of literals introduced to the formula by eliminating that variable.

3.4.2 Dependency graphs

One issue with prenex normal form is that much of the structural information of the original problem is lost on conversion. The quantifier prefix can be seen as a linear variable dependency scheme. In [Lonsing and Biere, 2010], the authors generalise variable dependency to directed acyclic graphs, which is more expressive and may more accurately represent the quantifier structure of the original problem. In a search based solver based on the extension of DPLL to QBF, variables are decided based on the partial ordering defined by the prefix. If the solver instead has access to the more general dependency graph it may have a greater degree of freedom with which to choose the order of decisions and maintain soundness.

Additionally, certain standard optimisations to the search procedure rely on dependency information for correctness. For instance a unit literal is the only unassigned existential literal l in a clause in which all unassigned universal literals are independent to l. A unit literal constrains the variable to the polarity of the literal and so decides that variable. With a more expressive dependency scheme it is possible to detect more unit literals and speed up the search.

3.4.3 Formula structure

Structural information about a formula can be used for other optimisations to QBF. Reconstructing a circuit similar to the original problem formulation can lead to a compressed representation and more efficient quantifier elimination [Pigorsch and Scholl, 2009, 2010]. The authors propose an and-inverter graph (AIG) representation and the application of circuit compression techniques such as BDD-sweeping for compression. BDDs may also be used to do quantifier elimination in cases where the representation does not explode. Otherwise quantification is performed directly on the AIG by symbolically expanding the circuit followed by compression.

Circuits may also be used as a representation of the problem in a search-based QBF solver [Goultiaeva et al., 2009]. The advantages in this setting

include propagation of assignments both forwards and backwards as well as identification of irrelevant *don't care* literals by analysing the gates of a circuit. This technique was further improved by the introduction of ghost literals [Klieber et al., 2010], which enables the solver to propagate cubes of learned satisfying assignments in the same way that learned constraints are.

3.4.4 SAT for QBF

SAT solvers are very efficient at finding satisfying assignment to existential queries but less efficient at proving unsatisfiability or, equivalently, satisfiability of universal queries. This asymmetry has been recognised and turned to an advantage in QBF solvers that use SAT to discover counterexamples to the universal component of QBF problems.

Counterexamples may be used to guide the careful expansion of a QBF [Janota and Marques-Silva, 2015] into a propositional formula. This algorithm provides the inspiration for the domain specific QBF solver in 4. The QBF is viewed as a game between an existential player and a universal player in which the existential player attempts to satisfy the formula and the universal player seeks to falsify it. The algorithm solves the game recursively by construction an abstraction, finding a candidate solution for one player under that abstraction, and subsequently verifying that candidate in the concrete game. The game is abstracted by partially expanding the QBF into propositional logic. A subset of possible assignments at each quantifier level are expanded into a disjunction for existential levels or conjunction for universal levels. If the current player cannot win in the abstraction against a restricted opponent then it cannot win in the concrete game so the abstraction can be used to find a candidate valuation to the player's current level variables with an efficient SAT call. The candidate is then checked by recursively calling the solver on the suffix of the formula. This effectively replaces the current player's choice with its candidate selection and hands control to the opponent. If the recursive call discovers a counterexample it is added to the abstraction and a new candidate is found. If there is no counterexample then the current player wins. If the refined abstraction now allows no candidate solution then the opponent wins. The full algorithm is listed in Algorithm 1.

Two recent counterexample guided approaches work on the idea of abstracting the QBF via the selection of a subset of clauses [Janota and Marques-Silva, 2015; Rabe and Tentrup, 2015]. In both works the authors suggest that

Algorithm 1 Counterexample guided QBF

```
1:  function SOLVE(QX(φ))
2:      if φ has no quantifiers then
3:          return (Q = ∃) ? SAT(φ) : SAT(¬φ)
4:      end if
5:      ω ← ∅
6:      loop
7:          α ← (Q = ∃) ? ⋀_{μ∈ω} φ[μ] : ⋁_{μ∈ω} φ[μ]
8:          τ′ ← SOLVE(PRENEX(QX(α)))
9:          if τ′ = NULL then return NULL
10:         τ ← {l | l ∈ τ′ ∧ var(l) ∈ X}
11:         μ ← SOLVE(φ[τ])
12:         if μ = NULL then return τ
13:         ω ← ω ∪ {μ}
14:     end loop
15: end function
```

competing SAT solvers select a subset of clauses for the opposing solver to satisfy at each quantifier alternation. In [Janota et al., 2012] the QBF abstraction is refined via expansion and may lead to an exponential increase in the size of the formula. By instead linearly increasing the formula to include selection variables enabling an abstraction over clauses the more recent approaches avoid that potential explosion.

An orthogonal approach uses nested SAT solvers to solve formulas of the form $\exists\sigma(\varphi \wedge (\neg\exists\tau(\psi)))$ where φ is a CNF, ψ is a QBF [Bogaerts et al., 2016]. At each quantifier level an underapproximation of ψ is given to a recursive solver while the CNF portion is solved via SAT. The SAT solver hands partial assignments gathered by propagating assignments through φ to the nested solvers. The partial assignment is validated on the underapproximation in order to discover conflicts from further inside the QBF.

3.5 Summary

The bounded realisability algorithm presented in this thesis can be seen as an extension of bounded model checking techniques into synthesis. Bounded realisability replaces the SAT calls in bounded model checking with QBF queries. In Chapter 4 I present a domain specific QBF algorithm that solves these queries more efficiently than a generalised QBF solver. In Chaper 5

and Chapter 6 I present extensions of the bounded realisability algorithm to synthesis and to unbounded games respectively. The approaches of those chapters are similar to existing extensions of bounded model checking into unbounded model checking. In this chapter we reviewed the literature on both bounded and unbounded model checking and the state of the art in QBF solving in order to show the inspirations for the work in this thesis.

This chapter also presented methodologies with a similar aim of applying the efficiency of SAT solving to synthesis. In the following chapters I will present my approach and defer a thorough comparison to the related methodologies until then.

4 Bounded Realisability

In this chapter I will describe my work on bounded realisability of reactive systems with safety properties. As introduced in Chapter 2 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 condition that defines a set of *error states* that the controller must avoid in order to be correct.

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 2 introduced games as a formalism for synthesis by stating the problem in terms of a game between a controller and its environment. In this chapter we are concerned with *bounded* games that restrict all runs in the game to certain length. This concept is borrowed from model checking where it is used to verify that a program emits no erroneous traces of a certain length. A propositional formula may be constructed that is satisfiable when a trace that visits an error state exists. A SAT solver can be used to efficiently search for a satisfying assignment to this formula, which represents a counterexample to the correctness property of the specification.

In realisability it is not enough to check for the existence of a trace that visits an error state. Such a trace only implies a counterexample that requires the controller to cooperate with the environment to fail. Instead we are interested in strategies for the players. A controller strategy must avoid the

error states for all possible environment actions. Likewise, a counterexample strategy must take into account all controller actions. We cannot use a SAT solver to search for a strategy directly as now we require quantifiers. We can, however, check if a strategy allows a counterexample trace without quantification. This sets us up for a counterexample guided methodology in which we construct candidate strategies and check them for correctness. If we discover a counterexample we use it to guide a refinement step in which we improve the candidate strategy.

Similar to bounded model checking, bounded realisability is not a complete procedure. If we decide that the controller can avoid error states for a game bounded to k rounds there is no guarantee that the environment can not force an error in a game with a bound higher than k. In Chapter 6 I present an extension to the algorithm that extends this algorithm to unbounded games.

Restricting ourselves to solving a bounded safety game enables us to turn the focus of the algorithm from states to traces. The traditional approach of constructing a binary decision diagram to symbolically represent the winning region has the potential to consume exponential space. The advantage of concentrating on runs of the game is that we do not rely on computing the winning states and therefore do not suffer from the related state explosion. The factors affecting the upper limit on scalability for the bounded synthesis algorithm are different to those of the BDD based approach. The most efficient algorithm for a realisability problem depends on the properties of that problem instance.

4.1 Algorithm

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. Quantifiers may be removed from a formula by iteratively constructing and merging copies of the formula for each quantified variable. The copies of the formula represent the two possible values, true or false, of the quantified boolean variable. Universal quantification can then be reduced to the conjunction of these copies, and existential quantifications corresponds to a disjunction. In practice, these

expanded formulas are far too large to be solved 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 3.

We define a safety game by the tuple $G = (S, \mathcal{U}, \mathcal{C}, \delta, s_0, E)$. $S, \mathcal{U}$, and $\mathcal{C}$ are sets of boolean variables representing game states, environment actions, and controller actions respectively. The transition relation, δ, is a boolean formula $2^S \times 2^{\mathcal{U}} \times 2^{\mathcal{C}} \rightarrow 2^S$ that maps current states and actions to successor states. The game begins in the initial state s_0 and the E is the set of error states that the controller must avoid.

The following quantified formula may be used to solve realisability of a safety game bounded to k game rounds:

$$\forall u_0 \exists c_0 ... \forall u_k \exists c_k \exists s_{0..k+1}$$
$$[\neg E(s_0) \wedge \delta(s_0, u_0, c_0, s_1) \wedge ... \wedge \neg E(s_k) \wedge \delta(s_k, u_k, c_k, s_{k+1}) \wedge \neg E(s_{k+1})]$$

The formula is constructed by unrolling the transition relationship for every game round until the bound is reached. A quantifier alternation is introduced to the formula for the variables corresponding to the actions of each player. Universal quantifiers are used for the environment variables and existential for the controller. The formula is constrained so that if the state at any game round is an error state the formula evaluates to false. In this way the formula is a satisfiable if and only if a strategy exists for the controller that avoids the error states.

It is possible to solve the QBF naïvely but we can do better by taking into account structural information in the realisability problem that is lost in the translation to prenex normal form. In particular, awareness that the formula is constructed of a repeated transition relation enables more effective learning. A naïve solver may learn sets of actions in order to prune the search tree. Instead we learn sets of states and, where possible, we transmit learned information between game rounds. For example, if we discover that the environment can force a visit to an error state from some set of states s in n rounds we also know that the environment can win from s in $n+m$ rounds for all $m > 1$. Knowledge of how to construct the formula also enables the solver to efficiently produce a smaller expanded formula when necessary by simply copying only the parts of the formula corresponding to game rounds after the action of the expanded quantifier.

4.1.1 Example

We introduce an example to assist an intuitive explanation of the algorithm. Consider a simple model of an ethernet device driver. The operating system makes requests of the driver to write or read data to or from the device. It is the role of the driver to grant these requests while ensuring that a `read` never occurs when a `write` was requests and vice versa.

As detailed in Chapter 2, we formalise realisability by a game structure $G = (S, \mathcal{U}, \mathcal{C}, \delta, s_0)$. The structure for our example is:

- $S = \{\texttt{request}, \texttt{error}\}$. The game consists of two boolean variables to denote the current request, and whether an error has occurred. We use `request = 0` to represent a `read` and `request = 1` for `write`.
- $\mathcal{U} = \{\texttt{os\_request}\}$. The uncontrollable actions consists of a single boolean variable to describe a `read` or a `write`. We use the same values as before, 0 for `read` and 1 for `write`.
- $\mathcal{C} = \{\texttt{dev\_cmd}\}$. The controllable actions similarly consists of a single boolean variable to denote the command given to the device: a `read` (`dev_cmd = 0`) or a `write` (`dev_cmd = 1`).
- The transition relation δ is defined by the following formula:

$$\texttt{request'} \leftarrow \texttt{os\_request}$$

$$\texttt{err'} \leftarrow \texttt{request} \neq \texttt{dev\_cmd}$$

  Primed variables are used here to indicate how the value is assigned in the next game round.

- $s_0 = \{\texttt{request = 0, err = 0}\}$. To simplify the example there is no idle state so the model is initialised with a pending `read` request.

The bounded synthesis algorithm is set within a counterexample guided abstraction refinement framework. An abstraction serves a dual purpose in this approach as both a representation of a player strategy and as a way to reduce the search space of the game. This is achieved by employing one player's candidate strategy as its opponent's game abstraction. The effect is

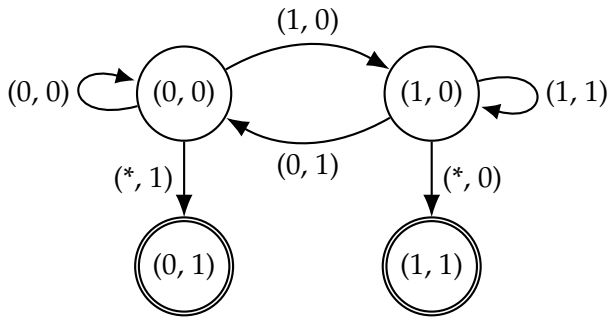
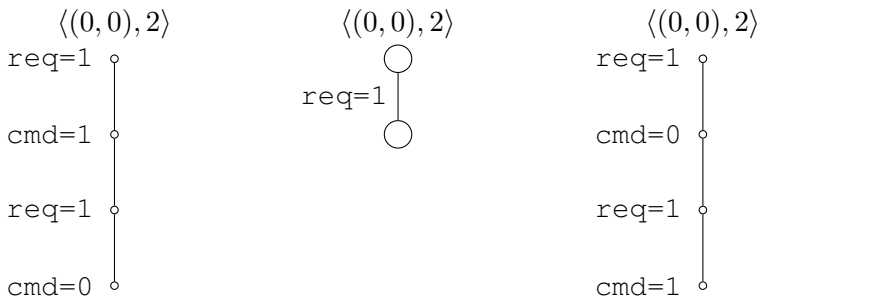


Figure 4.1: State automata representation of δ in Example 1. Nodes are labelled by the tuple (`request`, `error`). Edges are labelled with uncontrollable and controllable actions: (`os_request`, `dev_cmd`). A star indicates that the transition occurs on both a 0 and a 1. Transitions from error states are elided for simplicity.



(a) Trace to error (b) Candidate environment strategy (c) Counterexample trace

Figure 4.2: Execution of bounded realisability on the example.

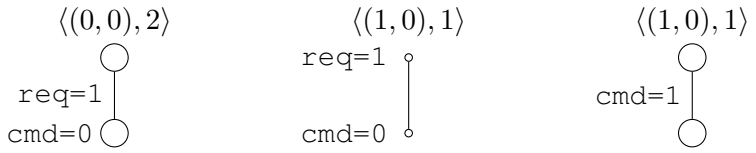
that the search for a player's strategy is directed by its opponent's current best effort strategy. Intuitively both players escalate their strategies until one of them converges on a winning strategy.

The abstractions of the game that we construct during the CEGAR search 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 **??** 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 **??** restricts the first environment action to `os_request=1`. At the leaf of the tree the game continues unrestricted.

We will now step through an execution of the algorithm using the example just introduced. The first step involves a search of the empty game abstraction for an initial candidate strategy for the environment player. In the empty abstraction we have not yet restricted the game in any way so all runs through the game are enabled. We search for a candidate strategy by finding a run that reaches an error state. Here we are only searching for the existence of a run and so we do not require quantifier alternations and a SAT solver can be used to efficiently perform the search. Intuitively, an existential search is equivalent to the two players of the game cooperating. In most real world applications of synthesis the environment is in fact cooperating with the system so the cooperation heuristic turns out to be practical. Figure 4.2a shows a trace through the example game that reaches an error state.

The trace informs us that by playing the actions contained in the trace it is possible for the environment to reach an error state. From this we conjecture that the first action in the trace is a reasonable choice for the environment's winning strategy. So we construct a candidate strategy in which the environment plays `os_request=1` in the first game round. The next step is to validate our conjecture by searching for counterexamples. We do this by constructing a new abstraction of the game in which the environment is restricted to playing actions from its candidate strategy (Figure 4.2b). Then we play this abstract game on the behalf of the controller. Once again we search for a trace through the game but this time the SAT solver is searching for a trace that avoids error states for the duration of the bounded game. Any traces found in this way indicate the possibility of a spoiling strategy for the controller that defeats the candidate strategy of the environment. Figure 4.2c shows a trace in which the controller counters the environment by playing `dev_cmd=0` in the first round to match the initial state of `request=0`.

The trace can be used to label nodes in the AGT representing a partial strategy for the controller for that abstract game (Figure 4.3a). Our goal now is to refine the candidate strategy for the environment so that it wins against the controller's partial strategy. In the candidate strategy we have not yet selected an environment action for the second game round, so we may refine the strategy by doing this now. Since the game is deterministic we can capture the actions in the AGT and the counterexample strategy with a single state: `{request=1, err=0}`. We now solve an unrestricted abstract game with a bound of 1 from this state in order to determine which action the environment



(a) Partial strategy (b) Trace to error (c) Refined AGT

Figure 4.3: Continued example algorithm execution

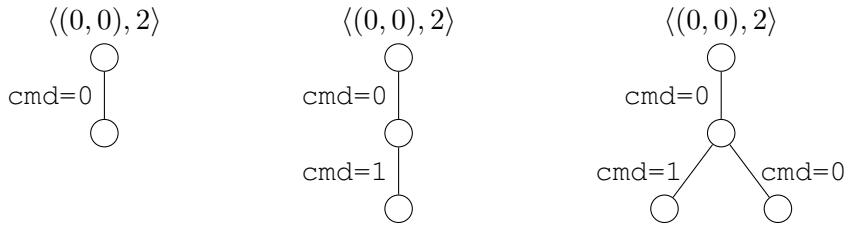
should select for the second game round.

The shorter game is solve via a recursive call to the algorithm. First a trace to an error state is found (Figure 4.3b) and a candidate constructed followed by a counterexample trace in which the controller chooses to correctly play `cmd=1`. At this point it is impossible to refine the environment candidate strategy by appending additional actions because the bound on game length has been reached. Instead an action from the candidate is backtracked and the search continues on a refined AGT that now includes the counterexample (Figure 4.3c).

The environment is now playing against a restricted opponent. In this case there is no possible action that the controller can take to reach an error state. This can be seen in the state machine for the game in Figure 4.1. If the environment cannot win from this state against a restricted controller then clearly it cannot win against an unrestricted controller in the game bounded to one round. We can therefore conclude that the candidate strategy that led to this state was not a winning strategy for its abstract game.

Actually we may conclude a stronger assertion that any strategy that results in this state with one game round remaining is a bad strategy. It is possible to exclude these strategies from future searches in an optimisation described in Section 4.2.1.

The algorithm now backtracks to the very beginning. We have determined that the candidate environment strategy in Figure 4.2b can be defeated by the partial controller strategy in Figure 4.3a. Now the abstraction of the game is refined to include the counterexample. The original empty abstraction refined with a single counterexample action is shown in Figure 4.4a. Note that a



(a) Refined AGT (b) Refined AGT (c) Refined AGT

Figure 4.4: Continued example algorithm execution

trace reaching an error in which the environment plays `os_request=1` is still possible in this abstract game. The algorithm without optimisation will consider this candidate again for the new abstract game. The result will be the same and refinement occurs again, except now the game is refined to include a counterexample action from the second round (Figure **??**). On this game abstraction the candidate is blocked because the trace must include `dev_cmd=1` in the second round.

As a result the environment must choose a different action for the first round of the game. A SAT query will reveal that `os_request=0` can lead to an error when the controller plays `dev_cmd=1` in the second round. However, this candidate can be defeated by the controller choosing `dev_cmd=0` instead. The algorithm will discover this and refine the abstraction again to include the counterexample. In refined abstract game (Figure 4.4c) the environment has no winning trace and therefore the algorithm terminates and returns realisable for the controller.

4.1.2 Abstract game trees

An abstract game tree (AGT) is a restricted version of the concrete game in which fewer actions are available. 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 NODES(T) the set of all nodes of a tree T, LEAVES(T) the subset of leaf nodes. For edge e, ACTION(e) is the action that labels the edge, and for node n, HEIGHT(k, n) is the distance 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* $Strat : \text{NODES}(T) \to \mathcal{C}$ ($Strat : \text{NODES}(T) \to \mathcal{U}$) labels each node n of the tree with the controller's (environment's) action to be played in the game round corresponding to $\text{HEIGHT}(k, n)$. Given a partial strategy $Strat$, we can map each leaf l of the abstract game tree to $\langle s', i' \rangle = \text{OUTCOME}(\langle s, i \rangle, Strat, l)$ 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.

4.1.3 Counterexample guided realisability

The bounded realisability algorithm constructs candidate strategies for one player that serve the dual purpose of game abstraction for its opponent. The algorithm begins by discovering a candidate for the environment. Next we must determine is there are counterexamples to the candidate. This step is executed by constructing an abstract game tree from the environment's candidate strategy and recursively invoking the algorithm on this new abstraction. The recursive call plays against the environment's strategy on behalf of the controller. Thus the algorithm can be seen as running two competing solvers, for the controller and for the environment. By symmetrically playing for both players achieve the goal of directing the search towards strong strategies and counterexamples.

The full procedure is illustrated in Algorithms 2 to 4. SOLVEABSTRACT 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 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

Algorithm 2 Solve an abstract bounded game

```
1: function SOLVEABSTRACT(p, s, k, T)
2:     cand ← FINDCANDIDATE(p, s, k, T)              ▷ Look for a candidate
3:     if k = 1 then return cand                      ▷ Reached the bound
4:     T' ← T
5:     loop
6:         if cand = NULL then      ▷ No candidate: return with no solution
7:             return NULL
8:         end if
9:         ⟨cex, l, u⟩ ← VERIFY(p, s, k, T, cand)          ▷ Verify candidate
10:        if cex = false then      ▷ No counterexample: return candidate
11:            return cand
12:        end if
13:        T' ← APPEND(T', l, u)              ▷ Refine T' with counterexample
14:        cand ← SOLVEABSTRACT(p, s, k, T')      ▷ Solve refined game tree
15:    end loop
16: end function
```

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.

Algorithm 3 Find a candidate strategy

```
17: function FINDCANDIDATE(p, s, k, T)
18:     T̂ ← EXTEND(T)                    ▷ Extend the tree with unfixed actions
19:     if p = cont then
20:         f ← TREEFORMULA(k, T̂)
21:     else
22:         f ← ¬TREEFORMULA(k, T̂)
23:     end if
24:     sol ← SAT(s(X_T̂) ∧ f)
25:     if sol = unsat then
26:         return NULL                                  ▷ No candidate exists
27:     else
28:         ▷ Return partial strategy for T
29:         return {⟨n, c⟩ | n ∈ NODES(T), c = SOL(n)}
30:     end if
31: end function
```

Algorithm 4 Verify a candidate strategy

```
32: function VERIFY(p, s, k, T, cand)
33:     for l ∈ leaves(gt) do
34:         ⟨k′, s′⟩ ← OUTCOME(s, k, cand, l)        ▷ Get bound and state at leaf
35:         if p = CONT then
36:             T′ ← ∅
37:         else
38:             T′ ← {cand(l)}
39:         end if
40:         ▷ Solve for the opponent
41:         a ← SOLVEABSTRACT(OPPONENT(p), s′, k′, T′)
42:         if a ≠ NULL then return ⟨true, l, a⟩        ▷ Return counterexample
43:     end for
44:     return ⟨false, ∅, ∅⟩                          ▷ There was no counterexample
45: end function
```

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 5.

The tree is first extended to the maximum bound with edges that are labeled with arbitrary opponent actions (Algorithm **??**, 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 5 Tree formulas for Controller and Environment

function TREEFORMULA(k, T)
  if HEIGHT$(k, T) = 0$ **then**
    return $\neg \mathrm{E}(X_T)$
  else
    return $\neg \mathrm{E}(X_T) \wedge$

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

  end if
end function
function $\overline{\text{TREEFORMULA}}(k, T)$
  if HEIGHT$(k, T) = 0$ **then**
    return $\mathrm{E}(X_T)$
  else
    return $\mathrm{E}(X_T) \vee$

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

  end if
end function

4.1.4 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 actions 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.

4.2 Optimisations

The bounded realisability algorithm has an exponential worst case running time when the entire search tree must be explored before discovering a winning strategy. In this section I present some optimisations that aim to prune the search tree as well as discover winning strategies earlier in the search.

4.2.1 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 6). 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 **??**.

Algorithm 6 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 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

4.2.2 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 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 7 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
```

4.2.3 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 FINDCANDIDATE. 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.

4.3 Discussion

The design of the algorithm is motivated by the desire to solve bounded safety games whilst avoiding the potential state explosion of computing the winning set. The key insight is to shift the emphasis from finding a winning set to finding winning strategies. The shift is made possible by searching for runs in an abstraction of the game and using the results to refine the abstraction. The advantage of this approach is that even when the winning set is difficult to represent symbolically (via a BDD or similar) a winning strategy may still be found. The reverse is also true: if the winning strategy requires too much branching it will become intractable to construct it using this algorithm. The difference can be likened to breadth-first versus depth-first search, the appropriate algorithm depends on the particular problem instance

to be solved.

4.3.1 Comparison to QBF

As previously stated the bounded realisability problem is a specialisation of QBF and the algorithm presented in this chapter is a domain specific QBF solver. In Chapter 3 several approaches to solving general QBF problems were described. The state of the art in QBF is focused on several areas: dependency analysis, circuit analysis, and counterexample guided search.

Analysing dependencies or the original circuit of the presented QBF are both aimed at reconstructing information that was lost during the conversion to a prenex normal form formula. The aim is to use that information to speed up the search for a solution. The bounded realisability algorithm is aware of the original problem formulation and constructs parts of it on the fly. For example, the quantifier dependency graph for a bounded realisability problem would normally be mostly linear due to the construction via unrollings of the transition relation, which rarely contains fully independent subformulas. An exception to this is the set of state and Tseitin variables which belongs to the last existential quantifier in the prefix. Those variables are decided by the action variables preceding them in the game. The algorithm presented here never includes anything other than action variables in player strategies, which reflects an awareness of that dependency.

The algorithm contains a higher level of information about the original problem than an analysis of the circuit could provide. It may be possible to implement some the propagation techniques introduced by these methods but in general learning sets of states provides a greater pruning of the search tree. We are able to transmit learned states between game rounds, which would not be possible in a general QBF solver.

The ability to learn states is also the primary difference between the specialised solver and the counterexample guided QBF algorithm proposed in [Janota et al., 2012]. More recent QBF approaches use counterexamples to guide an abstraction of the problem via clause selection. It would interesting to apply these techniques to bounded realisability although the extension to unbounded synthesis in Chapter 6 would not be applicable without and abstract game tree.

4.3.2 Model checking

The concept of solving games by searching for counterexamples of a certain length with a SAT solver was first introduced in a bounded approach to model checking [Biere et al., 1999]. Replacing counterexample traces with trees is the natural extension of this approach to realisability given that the controller now has agency to select actions in the game rather than modelling an implementation. The move from model checking to realisability brings additional complexity to the problem but the efficiency of SAT is still exploited to discover counterexamples quickly.

4.3.3 Related synthesis techniques

Bounded synthesis [Finkbeiner and Schewe, 2013] uses an SMT solver to search for a bounded *implementation* for an LTL specification. Lazy synthesis [Finkbeiner and Jacobs, 2012] similarly searches for a bounded partial implementation and uses BDD based model checking to search for counterexamples in order to refine the partial implementation. Both of these techniques are for full LTL synthesis, which is a different problem to the safety specifications solved in this chapter, but the overarching framework is similar.

Algorithms that avoid BDDs in favour of SAT solving have been proposed for safety synthesis in the past [Bloem et al., 2014; Chiang and Jiang, 2015; Morgenstern et al., 2013] but none of these take the approach of unrolling the transition relation to a bound. In these previous works states belonging to a winning region for one player are collected over a series of SAT queries concerning single transitions. These works have more in common with the extension of bounded realisability to unbounded games in Chapter 6.

4.3.4 Limitations

The performance of bounded realisability is influenced primarily on the branching factor of the game tree. The worst case scenario occurs when each environment action must be matched with a different controller action and no state learning is possible. For example, consider a modified version of the example given in Section 4.1.1 in which the environment requests are not latched into a state variable.

- $S = \{\texttt{error}\}$. The only state is whether or not an error has occurred.

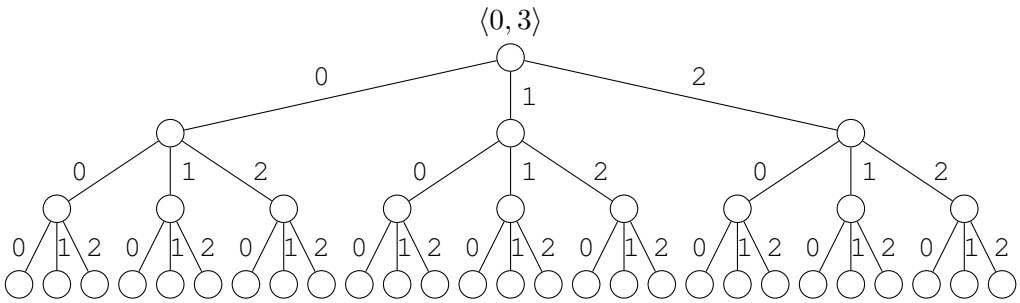


Figure 4.5: AGT with large branching factor

- $\mathcal{U} = \{\texttt{os\_request}\}$.
- $\mathcal{C} = \{\texttt{dev\_cmd}\}$.
- The transition relation δ is now:

$$\texttt{err'} \leftarrow \texttt{os\_request} \neq \texttt{dev\_cmd}$$

- $s_0 = \{\texttt{err = 0}\}$.

Without a state variable to learn the algorithm is forced to explore all possible actions to determine realisability. If the example is modified again to allow for more types of requests by increasing the size of the action variables then it is easy to see the potential blow up. In Figure 4.5 the final AGT for this example with a bound of 3 and action variables of size 3 is shown. The figure shows the values of `os_request` in all possible paths through the game. It is clear that this algorithm cannot scale with these kinds of specifications. It should be noted that it is trivial to solve this example with a BDD solver, which can immediately prove that `err = 0` is a winning region for the controller. The extension to unbounded realisability in Chapter 6 will also be able to handle this example by constructing the winning region.

4.3.5 Strengths

Bounded realisability is most useful in the case where the winning region of a game has a large BDD but the winning strategy for the game is compact. It is not fair to compare the incomplete bounded algorithm to a complete winning region computation but we may consider the ability for each technique to find counterexamples.

To demonstrate the usefulness of the algorithm we introduce a simple warehouse robot controller. In this example the warehouse consists of four loading bays and the robot is tasked with shipping items placed in the bays in a timely fashion. We model the problem with a `timer` variable that begins at one, ticks down to zero, and then resets back to one. At the beginning of the cycle the environment may items in any two bays. The robot then may ship all items in one bay per timer tick and must clear all bays before the timer resets. The example is trivial but it could be scaled on the number of bays, number of items that the environment can load, and the length of the timer to produce complex specifications. In this example we use integer values for `timer`, `ship`, `load0` and `load1` to make the description more concise.

- $S = \{$`error, bay0, bay1, bay2, bay3, timer`$\}$.
- $\mathcal{U} = \{$`load0, load1`$\}$.
- $\mathcal{C} = \{$`ship`$\}$.
- The transition relation δ is now:

$$
\begin{aligned}
\texttt{error'} &\leftarrow \texttt{timer=1} \wedge (\texttt{bay0=1} \vee \texttt{bay1=1} \vee \texttt{bay2=1} \vee \texttt{bay3=1}) \\
\texttt{bay0'} &\leftarrow (\texttt{timer=1} \wedge (\texttt{load0=0} \vee \texttt{load1=0})) \vee (\texttt{timer} \neq 1 \wedge \texttt{ship=0}) \\
\texttt{bay1'} &\leftarrow (\texttt{timer=1} \wedge (\texttt{load0=1} \vee \texttt{load1=1})) \vee (\texttt{timer} \neq 1 \wedge \texttt{ship=1}) \\
\texttt{bay2'} &\leftarrow (\texttt{timer=1} \wedge (\texttt{load0=2} \vee \texttt{load1=2})) \vee (\texttt{timer} \neq 1 \wedge \texttt{ship=2}) \\
\texttt{bay3'} &\leftarrow (\texttt{timer=1} \wedge (\texttt{load0=3} \vee \texttt{load1=3})) \vee (\texttt{timer} \neq 1 \wedge \texttt{ship=3}) \\
\texttt{timer'} &\leftarrow (\texttt{timer=0}) \texttt{ ? } 1 \texttt{ : } (\texttt{timer - 1})
\end{aligned}
$$

- $s_0 = \{$`error = 0, bay0 = 0, bay1 = 0, bay2 = 0, bay3 = 0, timer = 1`$\}$

The specification is clearly unrealisable given that in every cycle the environment can load items into two bays and the controller can only remove items from one bay. A realisability solver that uses BDDs might attempt to compute a winning region for the environment. The entire environment winning region for this game contains all possible configurations in which the environment loads two distinct bays. As shown in Figure 4.6, BDDs are not succinct on are in this enumeration of cubes style. If this problem was to scale the BDD would quickly consume a large amount of space.

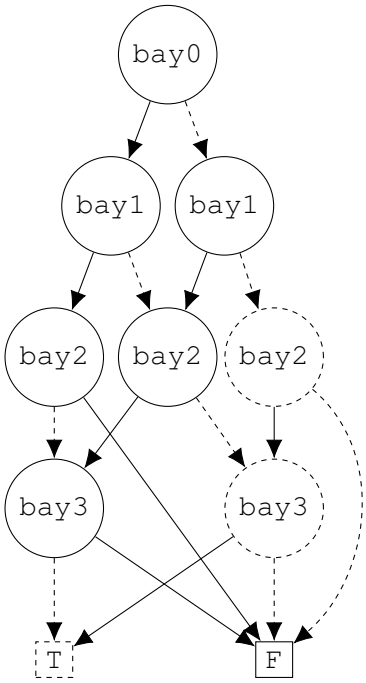


Figure 4.6: Environment winning region as a BDD.
Solid transitions are 1, dashed transitions are 0.

By instead using a SAT solver to check for the existence of a spoiling strategy to the environment filling two bays, the bounded realisability algorithm avoids computing the set of all environment winning states. The difference between the two methodologies is similar to the difference between breadth first search and depth first search. The BDD driven approach explores the game tree level by level similar to BFS and builds a compact set of winning states. The SAT based algorithm present here finds single paths through the tree that may be winning and checks them. Both approaches are efficient on different classes of specifications.

4.4 Summary

In this chapter I presented the fundamental building block of this thesis, a new algorithm for solving bounded realisability. In later chapters I will explain extensions to this algorithm to increase its applicability and in Chapter 7 I will present results and an evaluation of the contribution of this work.

- Here we introduce an algorithm for solving synthesis games that are

bounded to a fixed number of game rounds. The algorithm is a counterexample guided abstraction refinement framework in which abstractions of the game are constructed from candidate strategies for the players. This is done in a way that allows a candidate strategy to be checked for a spoiling strategy by playing the game abstraction on behalf of the opponent. Spoiling strategies are counterexamples to a strategy that may be used for refinement.

- The design of the algorithm is inspired by the exponential blow up that can result from constructing a symbolic representation of the winning region as a BDD. In this algorithm the winning region is never computed although some winning states are learned as an optimisation to prune the search tree. In Chapter 6 we will see how this algorithm may be extended to unbounded synthesis by approximating the winning region during the execution of the algorithm.

5 Strategy Extraction

In the previous chapter I introduced an algorithm for solving realisability for bounded safety games. In most applications of synthesis it is desirable to construct a controller strategy rather than merely prove its existence. In this chapter I will introduce a strategy extraction procedure that complements the bounded reachability algorithm. This process takes abstract game trees generated during reachability analysis and, using Craig interpolation, extracts mappings of states to player actions. By using interpolation this step can be done efficiently.

The problem solved in this chapter is related to the extraction of a Skolem function for a QBF. Recall from Chapter 2 that a Skolem function f provides a mapping from a prefix of universal variables $\hat{y}_0, \hat{y_1}, ..., \hat{y_i}$ to existential variables $\hat{x}$ such that when substituting $\hat{x}$ for $f(\hat{y}_0, \hat{y_1}, ..., \hat{y_i})$ the QBF is equisatisfiable. A Skolem function for a bounded realisability QBF gives a mapping from a prefix of past environment actions to a controller action, i.e. a strategy for that game round. A strategy for the entire game consists of a Skolem function for every round. In this chapter, we construct a strategy by replacing the prefix of previous environment actions with a game state and current environment action pair. This allows us to solve a simpler problem than Skolemisation of the entire QBF by once again taking advantage of the structure of the bounded realisability QBF.

5.1 Algorithm

Recall that a safety game is a tuple $(\mathcal{S}, \mathcal{U}, \mathcal{C}, \delta, s_0)$ where S is a set of states, $\mathcal{U}$ a set of environment action variables, $\mathcal{C}$ a set of controller action variables, δ defines a transition relation, and s_0 is an initial state. A set of states E

provides the winning condition, the controller must avoid error states and the environment must reach one. A winning strategy for controller is a function $\pi^c : 2^{\mathcal{S}} \times 2^{\mathcal{U}} \to 2^{\mathcal{C}}$ that avoids error states for the duration of the game. A controller strategy is then a mapping from state environment-action pairs to controller actions.

In this chapter we will assume that the safety game is winning for the controller. However, the technique is also easily applied to the environment to extract a spoiling strategy. In computing realisability of a safety game the algorithm constructs a *certificate tree*, which is an abstract game tree T such that for a set of states s and game bound κ, $s \wedge \overline{\text{TREEFORMULA}}(\kappa, \text{EXTEND}(T))$ is false. In other words it is a game abstraction for which the environment has no candidate strategy.

We use the certificate tree computed by the game solver as a starting point for strategy generation. We know that the controller can win the game in κ rounds by picking actions from the tree; however we do not yet know which action to choose in which situation.

5.1.1 Example

Figure 5.1 introduces the running example for this chapter. It shows a state machine for a game $(\mathcal{S}, \mathcal{U}, \mathcal{C}, \delta, s_0)$ that describes the operation of a simpled clocked flip-flop. For the example we model the clock as part of the game state and assume that it oscillates in every game round. So $\mathcal{S} = \{c, q, e\}$ is the state space of the game and contains the clock, the previous value of the flip-flop, and an error bit respectively. The environment has a single data bit variable: $\mathcal{U} = \{d\}$ and the controller has the next value of the flip-flop: $\mathcal{C} = \{q_n\}$. The diagram shows δ as a deterministic finite state automaton with edges labelled by a tuple (d, q_n). We use $*$ as a wildcard value to simplify the presentation. The circuit described by the specification allows the environment to save a data bit d into the flip-flop when the clock has a falling edge (c transitions from 1 to 0). The controller must correctly set q_n so that q always contains the correct data. The initial state, s_0, is $c = 0 \wedge q = 0 \wedge e = 0$ and the error set is $e = 1$.

The strategy generation algorithm takes an initial set s_0 and its certificate tree T, computed by the game solver, and generates a winning controller strategy starting from s_0. The generated strategy must define a controller action $c \in 2^{\mathcal{C}}$ to every $\{(s, u) \mid s \in 2^{\mathcal{S}}, u \in 2^{\mathcal{U}}\}$. The algorithm begins

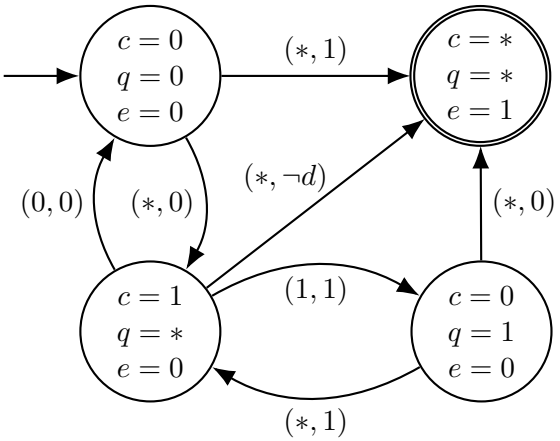
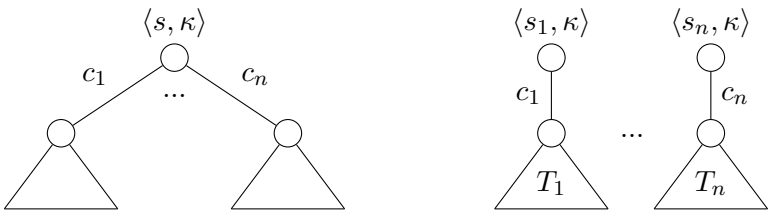


Figure 5.1: Transition relation of the running example

with $(s_0, \top)$, which it then partitions in pairs of subsets (s_i, u_i), one for each outgoing branch of T (Figure 5.2), such that the controller can win from s_i when the controller plays u_i by picking action c_i along the ith branch of the tree. This partitioning defines the local winning strategy in the root node of T. Next, for each partition (s_i, u_i), the algorithm computes the set of c_i-successor states of s_i, obtaining the set s'_i of states winning in the child subtree T'_i of T_i (Figure 5.2). The algorithm is then invoked recursively for each subtree T'_i.

Figure 5.3 illustrates operation of the algorithm on the winning abstract game tree returned by the game solver for our running example. Figure 5.3a displays T, the certificate tree for the game. The algorithm starts at the root of the tree and the initial set $s = (c = 0 \wedge q = 0 \wedge e = 0)$. The game tree defines only one winning action in the root node, hence this action is winning in all states of s and no partitioning is required. We compute the successor set reachable by playing action $q_n = 0$ in s: $\{s' \mid \delta(s, \top, q_n = 0, s')\} = (c = 1 \wedge e = 0)$.



(a) Before (b) After

Figure 5.2: Partitioning

Next, we descend down the tree and consider subtree T' and its initial set s' (Figure 5.3b). We partition $(s', \top)$ into subsets $(s'_1, u'_1) \leftarrow (c = 1 \wedge e = 0, d = 0)$ and $(s'_1, u'_1) \leftarrow (c = 1 \wedge e = 0, d = 1)$ that are winning for the left and right subtrees of T' respectively, i.e., from $(c = 1 \wedge e = 0)$ the controller must play action $q_n = 0$ when the environment plays $d = 0$, and $q_n = 1$ for $d = 1$. Consider the resulting subtrees T'_1 and T'_2 with initial sets s'_1 and s'_2 (Figure 5.3c). We have $\{s''_1 \mid \delta(s'_1, u'_1, q_n = 0, s''_1)\} = (c = 1 \wedge q = 0 \wedge e = 0)$, $\{s''_2 \mid \delta(s'_2, u'_1, q_n = 1, s''_2)\} = (c = 1 \wedge q = 1 \wedge e = 0)$. Finally, we obtain two subtrees T''_1 and T''_2 with initial sets s''_1 and s''_2 (Figure 5.3d). Both subtrees have one branch; hence corresponding actions $q_n = 0$ and $q_n = 1$ are winning for $(s''_1, \top)$ and $(s''_2, \top)$ respectively.

Putting together fragments of the winning strategy computed above, we obtain the following strategy for this example:

$$\begin{aligned}
\pi(c = 0 \wedge q = 0 \wedge e = 0, \top) &= (q_n = 0) \\
\pi(c = 1 \wedge e = 0, d = 0) &= (q_n = 0) \\
\pi(c = 1 \wedge e = 0, d = 1) &= (q_n = 1) \\
\pi(c = 0 \wedge q = 1 \wedge e = 0, \top) &= (q_n = 1)
\end{aligned}$$

The above algorithm involves two potentially costly operations: winning

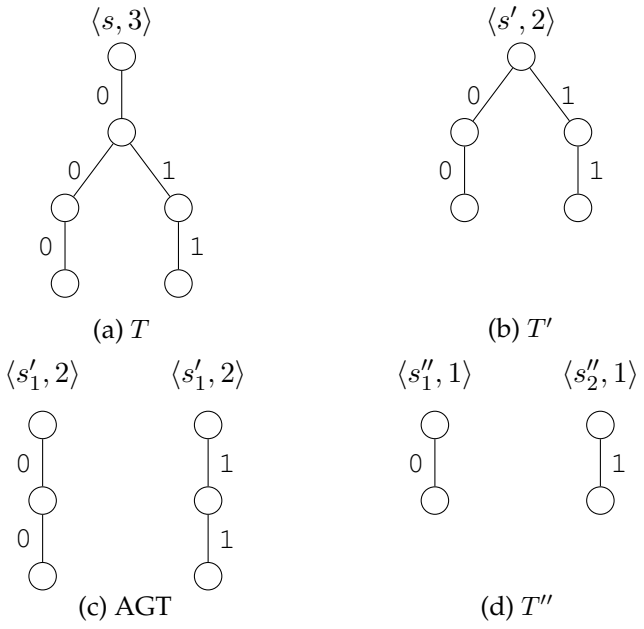


Figure 5.3: Operation of the strategy extraction algorithm on the example

set partitioning and successor set computation. If implemented naïvely, these operations can lead to unacceptable performance. The key insight behind our solution is that both operations can be efficiently approximated from the proof of unsatisfiability of the formula $s \wedge \overline{\text{TREEFORMULA}}(T)$, with the help of interpolation, as described below. The resulting approximations are sound, i.e., preserve the correctness of the resulting strategy.

5.1.2 Local Strategies

Algorithm 8 shows the pseudocode of the strategy generation algorithm. The algorithm proceeds in two phases: the first phase (GENLOCALSTRATS) computes local strategies in nodes of T; the second phase (COMPILESTRAT) compiles all local strategies into a winning strategy function.

The GENLOCALSTRATS function recursively traverses the certificate tree T, starting from the root, computing local strategies in each node. The main operation of the algorithm, called PARTITION, splits $(T, s, \top)$ into j tuples (T_i, s_i, u_i), as shown in Figure 5.2. Each tree T_i is a copy of a single branch of T. The partitioning is constructed in such a way that the action c_i that labels the root edge of T_i is a winning controller action in s_i against environment action u_i.

Next we consider each tuple (T_i, s_i, u_i) (lines 9-13). We descend down the tree and compute the controller strategy in the child subtree T_i' of T_i (right-hand side of Figure 5.2). To do so, we first compute the set of c_i-successors of (s_i, u_i): More precisely, we compute an overapproximation $\mathcal{I} \supseteq \{s_i' \mid \delta(s_i, u_i, c_i, s_i')\}$, such that T_i' is a certificate tree for $\mathcal{I}$. Such an overapproximation is returned by the NEXT function in line 10. We can now recursively invoke the strategy generation function to compute a winning strategy for the subtree T_i' from $\mathcal{I}$ (line 11).

The algorithm returns the set of tuples (W, c, k). Each tuple represents a fragment of the strategy in some tree node, where $W \in 2^{\mathcal{S}} \cup 2^{\mathcal{U}}$ is the winning set in this node, c is the controller action to play in this set, and k is the distance from the node to the bottom of the tree.

5.1.3 Partitioning game trees

The PARTITION function (Algorithm 9) computes a local strategy in the root of an abstract game tree. It takes a pair (T, s), such that T is a certificate tree for

Algorithm 8 Computing a winning strategy

function GENSTRATEGY(T, s)
  $Strat \leftarrow$ GENLOCALSTRATS(T, s)
  return COMPILESTRAT$(Strat)$
end function

function GENLOCALSTRATS(T, s)
  $[(e_1, n_1), \ldots, (e_j, n_j)] \leftarrow$ SUCC(T)
  $[(T_1, s_1, u_1), \ldots, (T_j, s_j, u_j)] \leftarrow$ PARTITION$(T, s \wedge \neg E(\mathcal{S}_T))$
  $Strat \leftarrow \{(s_i, u_i, \text{ACTION}(e_i), \text{HEIGHT}(T)) \mid i \in [1, .., j]\}$
  for $i = 1$ to j **do**
    $(T_i', s_i') \leftarrow$ NEXT(T_i, s_i, u_i)
    $Strat_i \leftarrow$ GENLOCALSTRATS(T_i', x_i')
    $Strat \leftarrow Strat \cup Strat_i$
  end for
  return $Strat$
end function

Algorithm 9 Partitioning winning states

function PARTITION(T, s)
  $\hat{s} \leftarrow s$
  $\hat{u} \leftarrow \top$
  $\hat{T} \leftarrow T$
  for $i = 1$ to j **do**
    $(T_i, \tilde{T}) \leftarrow$ SPLIT$(\hat{T})$
    $A \leftarrow \hat{s} \wedge \hat{u} \wedge \overline{\text{TREEFORMULA}}(\tilde{T})$
    $B \leftarrow \overline{\text{TREEFORMULA}}(T_i)$
    $\mathcal{I} \leftarrow$ INTERPOLATE(A, B)
    $(s_i, u_i) \leftarrow (\mathcal{I}(\mathcal{S}_T) \wedge \hat{s}, \mathcal{I}(\mathcal{U}_T) \wedge \hat{u})$
    $\hat{s} \leftarrow \hat{s} \wedge \neg s_i$
    $\hat{u} \leftarrow \hat{u} \wedge \neg u_i$
    $\hat{T} \leftarrow \tilde{T}$
  end for
  return $[(T_1, s_1, u_1), \ldots, (T_j, s_j, u_j)]$
end function

set s and partitions s into subsets (s_i, u_i) such that the controller can win in s_i when the opponent plays u_i by choosing action c_i.

At every iteration, the algorithm splits the tree into the leftmost branch T_i and the remaining tree (Figure 5.4). It then computes the pair (s_i, u_i) where the controller wins by following the branch T_i, removes s_i from the initial set

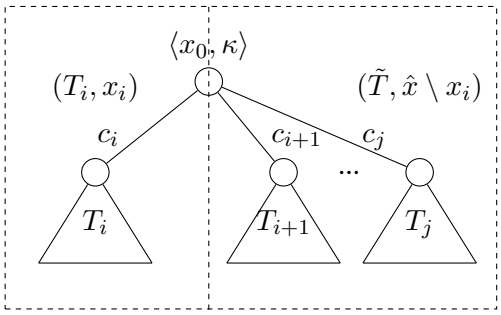


Figure 5.4: Splitting of T in the PARTITION function.

s, and u_i from the set of opponent moves u. At the next iteration it considers the leftover tree $\tilde{T}$ and shrunk initial sets $\hat{s}$ and $\hat{u}$.

The algorithm maintains the invariant that $\hat{T}$ is a certificate tree for $\hat{s}$ if the environment chooses from $\hat{u}$, and hence $\hat{s} \wedge \hat{u} \wedge \overline{\text{TREEFORMULA}}(\hat{T})$ is unsatisfiable. We decompose this formula into two conjuncts $A \wedge B$ such that A and B only share state and action variables $\mathcal{S}_T \cup \mathcal{U}_T$ in the root node of T and that the interpolant $\mathcal{I}$ of A and B consists of states and environment actions for which the controller can win by following the T_i subtree. Hence $(\mathcal{I}(\mathcal{S}_T) \wedge \hat{s}, \mathcal{I}(\mathcal{U}_T) \wedge \hat{u})$ gives us the desired pair (s_i, u_i).

Informally, A is a partial expansion of the game formula induced by $\tilde{T}$. It is satisfiable iff there exists a spoiling environment strategy from $\hat{s}$ and allowable by $\hat{u}$ against abstract game tree $\tilde{T}$. B is a partial expansion of the game induced by T_i. It is satisfiable iff there exists a spoiling environment strategy against T_i. Both A and B can be satisfiable individually, but because T is a certificate tree their conjunction is unsatisfiable.

The interpolant $\mathcal{I}$ of A and B implies $\neg B$, i.e., for any state and environment action in $\mathcal{I}$, c_i is a winning move. $\mathcal{I}$ is also implied by A, i.e., it contains all states in s and all environment actions for which the controller cannot win by picking moves from $\tilde{T}$ as a subset. Equivalently, for any state in $\hat{s} \wedge \neg\mathcal{I}(\mathcal{S}_T)$ and environment action in $\hat{u} \wedge \neg\mathcal{I}(\mathcal{U}_T)$, the controller *can* win by following $\tilde{T}$, i.e., $\tilde{T}$ is a certificate tree for $(\hat{s} \wedge \neg\mathcal{I}(\mathcal{S}_T), \hat{u} \wedge \neg\mathcal{I}(\mathcal{U}_T))$, and we can apply the decomposition again to $\tilde{T}$ at the next iteration.

We prove useful properties of the PARTITION function. We begin with the proposition that A and B imply a decomposition of $\hat{s} \wedge \hat{u} \wedge \overline{\text{TREEFORMULA}}(\hat{T})$.

Proposition 1. $A \wedge B \implies \hat{s} \wedge \hat{u} \wedge \overline{\text{TREEFORMULA}}(\hat{T})$.

Proof.

$$
\begin{aligned}
A \wedge B &= (\hat{s} \wedge \hat{u} \wedge \overline{\textsc{treeFormula}}(\tilde{T})) \wedge \overline{\textsc{treeFormula}}(T_i) \\
A \wedge B &= \hat{s} \wedge \hat{u} \wedge \Bigg(E(\mathcal{S}_{\tilde{T}}) \vee \\
&\qquad \bigvee_{\langle e,n\rangle \in \textsc{succ}(\tilde{T})} \delta(\mathcal{S}_{\tilde{T}}, \mathcal{U}_{\tilde{T}}, \mathcal{C}_{\tilde{T}}, \mathcal{S}_n) \wedge \textsc{action}(e) \wedge \overline{\textsc{treeFormula}}(n) \Bigg) \\
&\quad \wedge \Bigg(E(\mathcal{S}_{T_i}) \vee (\delta(\mathcal{S}_{T_i}, \mathcal{U}_{T_i}, \mathcal{C}_{T_i}, \mathcal{S}_{n_i}) \wedge \textsc{action}(e_i) \wedge \overline{\textsc{treeFormula}}(n_i)) \Bigg) \\
\implies \hat{s} \wedge \hat{u} \wedge &\Bigg(E(\mathcal{S}_{\hat{T}}) \vee \\
&\qquad \bigvee_{\langle e,n\rangle \in \textsc{succ}(\hat{T})} \delta(\mathcal{S}_{\hat{T}}, \mathcal{U}_{\hat{T}}, \mathcal{C}_{\hat{T}}, \mathcal{S}_n) \wedge \textsc{action}(e) \wedge \overline{\textsc{treeFormula}}(n) \Bigg) \\
&= \hat{s} \wedge \hat{u} \wedge \overline{\textsc{treeFormula}}(\hat{T})
\end{aligned}
$$

□

Proposition 2. *The following invariant is maintained throughout the execution of* PARTITION*: $\hat{T}$ is a certificate tree for $\hat{s}$ when the environment may choose actions from $\hat{u}$.*

Proof. We prove by induction. It is a precondition of the function that T is a certificate tree for s, thus the invariant holds for the initial values of $\hat{T} = T$, $\hat{s} = s$, and $\hat{u} = \top$. By the induction hypothesis $(\hat{s} \wedge \hat{u} \wedge \overline{\textsc{treeFormula}}(\hat{T}))$ is unsatisfiable, so by Proposition 1 $(A \wedge B)$ must also be unsatisfiable. Hence the interpolation operation in line 9 is well defined. By the properties of interpolants, $(A \implies \mathcal{I})$, hence $(\neg\mathcal{I} \implies \neg A)$ or equivalently $(\neg\mathcal{I} \implies \neg(\hat{s} \wedge \hat{u} \wedge \overline{\textsc{treeFormula}}(\tilde{T}))$.

After $\hat{T}$, $\hat{s}$, and $\hat{u}$ are updated in line 13, their new values $\hat{T}'$, $\hat{s}'$, and $\hat{u}'$ satisfy the following equalities:

$$
\begin{aligned}
\hat{s}' \wedge \hat{u}' \wedge \overline{\textsc{treeFormula}}(\hat{T}') &= \hat{s} \wedge \hat{u} \wedge \overline{\textsc{treeFormula}}(\tilde{T}) \wedge \neg\mathcal{I} \\
&= \neg\mathcal{I} \wedge \hat{s} \wedge \hat{u} \wedge \overline{\textsc{treeFormula}}(\tilde{T}) \\
&\implies \neg(\hat{s} \wedge \hat{u} \wedge \overline{\textsc{treeFormula}}(\tilde{T})) \\
&\qquad \wedge \hat{s} \wedge \hat{u} \wedge \overline{\textsc{treeFormula}}(\tilde{T}) \\
&= \bot
\end{aligned}
$$

and hence the invariant is maintained. □

Proposition 3. *Let T be a certificate tree for s and let $s \wedge \neg E(\mathcal{S}_T) = \bot$. Then $[(T_1, s_1, u_1), \ldots, (T_j, s_j, u_j)] = \text{PARTITION}(T, s)$ is a local winning strategy in the root of T, i.e., the following properties hold:*

1. *Sets $(s_1, u_1), \ldots, (s_j, u_j)$ comprise a partitioning of $(s, \top)$: $s = \bigvee s_i$, $\top = \bigvee u_i$ and $\forall i, k.(i \neq k) \implies (s_i \wedge u_i) \wedge (s_k \wedge u_k) = \bot$*
2. *T_i is a certificate tree for s_i under environment actions u_i, for all $i \in [1, j]$*

Proof. At every iteration of the algorithm, we partition $\hat{s}$ into $s_i = \mathcal{I}(\mathcal{S}_T) \wedge \hat{s}$ and $\hat{s} \wedge \neg\mathcal{I}(\mathcal{S}_T)$. We do the same for $\hat{u}$: $u_i = \mathcal{I}(\mathcal{U}_T) \wedge \hat{u}$ and $\hat{u} \wedge \neg\mathcal{I}(\mathcal{U}_T)$. Hence the different sets in each s_i and u_i do not overlap by construction.

At the final iteration of the algorithm, the tree $\tilde{T}$ consists of a single root node without outgoing branches. Hence, $A = \hat{s} \wedge \hat{u} \wedge \overline{\text{TREEFORMULA}}(\tilde{T}) = \hat{s} \wedge \hat{u} \wedge \neg E(\mathcal{S}_{\tilde{T}}) = \hat{s} \wedge \hat{u}$. Since $(A \implies \mathcal{I})$, we get $((\hat{s} \wedge \hat{u}) \implies \mathcal{I})$ and therefore $\mathcal{I} \wedge \hat{u} \wedge \hat{s} = \hat{s} \wedge \hat{u}$, i.e., all states in $\hat{s}$ and $\hat{u}$ are included in the final pair (s_j, u_j) and hence the partitioning completely covers sets $(s, \top)$: $s = \bigvee s_i$ and $\top = \bigvee u_i$.

We prove the second statement of the proposition. The sets s_i and u_i are computed as $\mathcal{I}(\mathcal{S}_T) \wedge \hat{s}$ and $\mathcal{I}(\mathcal{U}_T) \wedge \hat{u}$ at the ith iteration of the algorithm (line 10). Thus,

$$\begin{aligned} s_i \wedge u_i \wedge \overline{\text{TREEFORMULA}}(T_i) &= \mathcal{I}(\mathcal{S}_T) \wedge \hat{s} \wedge \mathcal{I}(\mathcal{U}_T) \wedge \hat{u} \wedge \overline{\text{TREEFORMULA}}(T_i) \\ &= \mathcal{I} \wedge \hat{s} \wedge \hat{u} \wedge \overline{\text{TREEFORMULA}}(T_i) \end{aligned}$$

By the properties of interpolants, $\mathcal{I} \wedge B = \mathcal{I} \wedge \overline{\text{TREEFORMULA}}(T_i) = \bot$. Hence $s_i \wedge u_i \wedge \overline{\text{TREEFORMULA}}(T_i) = \bot$, i.e., T_i is a certificate tree for s_i under opponent actions u_i. □

5.1.4 Determine an action

The NEXT function (Algorithm 10) takes a set s, a set u, and its certificate tree T, such that there is exactly one outgoing edge, labelled c, from the root node of T. T has a sole child subtree T' with root node n. The function computes an overapproximation s' of the c-successor of (s, u), such that s' is winning for the controller and T' is a certificate tree for s'.

Once again, we decompose the unsatisfiable formula $s \wedge u \wedge \overline{\text{TREEFORMULA}}(T)$ into two conjuncts A and B. A encodes one round of the game from the set s under u, where the controller plays action c. $B = \overline{\text{TREEFORMULA}}(T')$ is

Algorithm 10 Successor set

1: **function** NEXT(T, s, u)
2: $[(e, n)] \leftarrow \text{SUCC}(T)$ ▷ T has a single successor
3: $A \leftarrow s \wedge u \wedge \delta(\mathcal{S}_T, \mathcal{U}_T, \mathcal{C}_T, \mathcal{S}_n) \wedge \text{ACTION}(e)$
4: $B \leftarrow \overline{\text{TREEFORMULA}}(n)$
5: $\mathcal{I} \leftarrow \text{INTERPOLATE}(A, B)$
6: **return** $(n, \mathcal{I}(\mathcal{S}_n))$
7: **end function**

a partial $\forall$-expansion of the game induced by T'. A and B only share state variables $\mathcal{S}_n$ (where n is the root node of T' and single successor node of T) and their interpolant gives the approximation we are looking for.

Proposition 4. *Let T be a certificate tree for s under opponent actions u with a single outgoing edge, labelled c in its root node, and let $(T', \mathcal{I}) = \text{NEXT}(T, s, u)$. Then:*

1. *$\mathcal{I}$ is an overapproximation of the c-successor of s under u, i.e.,*

$$\mathcal{I} \supseteq \{s' \mid \delta(s, u, c, s')\}$$

2. *T' is a certificate tree for s'*

Proof. The c-successor set s' of s under environment actions u is defined by $\exists s_T, u_T(s \wedge u \wedge \delta(s_T, u_T, c_T, s') \wedge c_T = c)$. The matrix of this formula is exactly formula A. Hence the successor set is given by $s' = \exists s_T, u_T(A)$. Since $(A \implies \mathcal{I})$, $s' \implies \exists s, u.\mathcal{I}$. Since $\mathcal{I}$ is defined over state variables in the root of T' only, the quantifiers can be removed: $s' \implies \mathcal{I}$ or, in the relational form, $\mathcal{I} \supseteq s'$.

We prove the second property by the construction of the interpolant: $(\mathcal{I} \wedge \overline{\text{TREEFORMULA}}(T')) = (\mathcal{I} \wedge B) = \bot$. □

5.1.5 Compiling the strategy

Finally, we describe how local strategies computed by GENLOCALSTRATS are combined into a winning strategy for the game. This requires some care, as individual partial strategies can be defined over overlapping sets of states. We want the resulting strategy function to be deterministic; therefore for each partial strategy we only add new states not yet covered by the computed combined strategy. Function COMPILESTRATS (Algorithm 11) achieves this

Algorithm 11 Compiling the winning strategy

```
1: function COMPILESTRAT(Strat)
2:     π ← ⊥,  W ← ⊥
3:     for (w, c, k) ∈ Strat do
4:         π ← π ∨ (w ∧ ¬W ∧ (C = c))
5:         W ← W ∨ w
6:     end for
7:     return π
8: end function
```

by keeping track of all states W already added to the strategy. For every new tuple (w, a, k), it restricts the set w to $\neg W$, which guarantees that no state action pair can be added to the strategy twice.

5.2 Related work

All existing strategy extraction algorithms for games were developed for use with game solvers based on winning set compilation [Bloem et al., 2014]. Such a solver generates a sequence of expanding state sets for which the game is safe for $1, 2, \ldots$ steps. The task of the strategy extraction algorithm is to compute a function that in every winning state chooses a single move that forces the game back into a safe state. In contrast, our strategy generation algorithm does not require the game solver to compile winning regions, but instead uses abstract game trees.

Another line of related work is strategy extraction algorithms for QBF used in QBF certification. QBF strategy extraction methods are specific to the underlying proof system used by the QBF search algorithm [Egly et al., 2013; Goultiaeva et al., 2011; Lonsing and Biere, 2010]. A strategy in a QBF is an oracle that, given the history of moves played in the game, outputs the next move for the winning player. An additional procedure is required to convert this oracle into a memory-free strategy function that maps a state to a controller move. Our work can been seen as such a procedure for $\forall Exp + Res$ proof system based solvers [Janota and Marques-Silva, 2013].

5.3 Summary

I have presented a strategy extraction algorithm to extended bounded realisability to bounded synthesis. The algorithm assigns controller actions to pairs of states and environment actions by a lightweight recursive application of interpolation to the certificate tree generated by the game solver. The performance overhead of this approach will be evaluated in Chapter 7.

6 Unbounded Realisability

In previous chapters I outlined an algorithm to solve bounded realisability games and an extension that can extract strategies from the result. Bounded realisability can be used to prove the existence of a winning strategy for the environment on the unbounded game by providing a witness. For the controller, the strongest claim that can be made is that the strategy is winning as long as the game does not extend beyond the maximum bound. The work described in this chapter can be used to address this by presenting another extension to the algorithm that solves unbounded realisability games.

The baseline solution to this problem is to set a maximum bound such that all runs in the unbounded game will be considered. The naïve approach is to use size of the state space as the bound ($2^{\mathcal{S}}$) so that all states may be explored by the algorithm. A more nuanced approach is to use the diameter of the game [Biere et al., 1999], which is the smallest number d such that for any state x there is a path of length $\leq d$ to all other reachable states. Computing the diameter, however, is also expensive and solving a game bounded to the size of the diameter may be infeasible.

Instead I present an approach that iteratively solves games of increasing bound while learning bad states from abstract games using Craig interpolation. We utilise the approximation properties of the interpolant to construct sets of states that underapproximate the total losing set for the controller. By underapproximating we avoid constructing a potentially large representation of this set that could be the cause of infeasibility in a BDD solver. Later in this chapter we will see that a careful construction of approximate sets enables a fixed point that is sufficient to prove the nonexistence of an environment-winning strategy.

6.1 Algorithm

Recall the bounded realisability algorithm from Chapter 4. A safety game is solved with respect to a bound on the number of game rounds. The algorithm is a counterexample guided approach that constructs abstractions of the game in the form of game trees. An abstract game tree restricts one player to the actions that label edges in the tree. In Chapter 5 I presented a strategy extraction procedure that utilises the abstract game tree as a certificate for the game. In this chapter I introduce a similar approach that extracts an approximation of winning states from the certificate tree.

The bounded algorithm is reproduced in Algorithm 12 with some modifications. The algorithm solves a game $(\mathcal{S}, \mathcal{U}, \mathcal{C}, \delta, s_0)$ with state variables $\mathcal{S}$, environment variables $\mathcal{U}$, controller variables $\mathcal{C}$, transition relation δ, and initial states s_0. The safety condition of the game is defined by a set of error states E that the controller must avoid and the environment must reach. The unbounded algorithm is modified to learn states when a candidate cannot be found for an abstract game (line 19).

We extend the bounded synthesis algorithm to learn states losing for one of the players from failed attempts to find candidate strategies. The learning procedure kicks in whenever FINDCANDIDATE cannot find a candidate strategy for an abstract game tree. We can learn additional losing states from the tree via interpolation. This is achieved in line 19 in Algorithm 12, enabled in the unbounded version of the algorithm, which invokes LEARN or $\overline{\text{LEARN}}$ to learn controller or environment losing states respectively (Algorithm 13).

6.1.1 Example

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 6.1 shows the variables (6.3a), the initial state of the system (6.3b), and the formulas for computing next-state variable assignments (6.3c) for this example. We use primed identifiers to denote next-state variables and curly braces to define the domain of a variable.

Algorithm 12 Unbounded realisability

```
1:  function SOLVEABSTRACT(p, s, k, T)
2:      cand ← FINDCANDIDATE(p, s, k, T)
3:      if k = 1 then return cand
4:      T' ← T
5:      loop
6:          if cand = NULL then return NULL
7:          ⟨cex, l, u⟩ ← VERIFY(p, s, k, T, cand)
8:          if cex = false then return cand
9:          T' ← APPEND(T', l, u)
10:         cand ← SOLVEABSTRACT(p, s, k, T')
11:     end loop
12: end function
13: function FINDCANDIDATE(p, s, k, T)
14:     T̂ ← EXTEND(T)
15:     f ← if p = cont then TREEFORMULA(k, T̂) else ¬TREEFORMULA(k, T̂)
16:     sol ← SAT(s(S_T̂) ∧ f)
17:     if sol = unsat then
18:         ▷ Unbounded solver learns states here
19:         if p = cont then LEARN(s, T̂) else ¬LEARN(s, T̂)
20:         return NULL
21:     else
22:         return {⟨n, c⟩ | n ∈ NODES(T) , c = SOL(n)}
23:     end if
24: end function
25: function VERIFY(p, s, k, T, cand)
26:     for l ∈ leaves(gt) do
27:         ⟨k', s'⟩ ← OUTCOME(s, k, cand, l)
28:         if p = CONT then
29:             T' ← ∅
30:         else
31:             T' ← {cand(l)}
32:         end if
33:         a ← SOLVEABSTRACT(OPPONENT(p), s', k', T')
34:         if a ≠ NULL then return ⟨true, l, a⟩
35:     end for
36:     return ⟨false, ∅, ∅⟩
37: end function
```

Uncontrollable	Controllable	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;
```

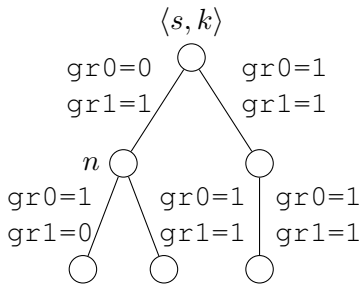
(c) Transition Relation

Figure 6.1: Example

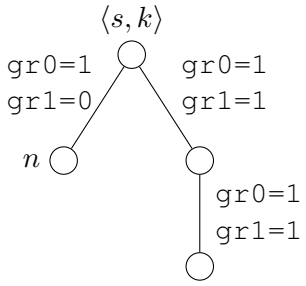
This example is the $n = 2$ instance of the more general problem of an arbiter of n resources. For large values of n, the set of winning states has no compact representation, which makes the problem hard for BDD solvers. One approach is to compute the uncontrollable predecessor of the set of error states, i.e. the states from which the environment can force the game into an error state. In the example, one iteration of this operation would give the set: $\texttt{nrequests} = 3 \vee (\texttt{nrequests} = 2 \wedge (\texttt{resource0} = 0 \vee \texttt{resource1} = 0)) \vee (\texttt{nrequests} = 1 \wedge (\texttt{resource0} = 0 \wedge \texttt{resource1} = 0))$. We try to avoid computing the entire winning set by employing interpolation to approximate it instead. The benefit of interpolation is that an approximation can be obtained efficiently from the resolution proof of two mutually unsatisfiability formulas.

Consider node n in Figure 6.2a, which shows an abstract game tree for which the environment has no winning action. At this node there are two controller actions that prevent the environment from forcing the game into an error state in one game round. We want to use this tree to learn the states from which the controller can win playing one of these actions.

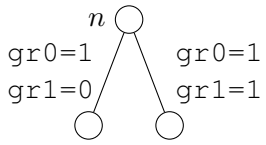
Given two formulas F_1 and F_2 such that $F_1 \wedge F_2$ is unsatisfiable, it is possible to construct a Craig interpolant $\mathcal{I}$ such that $F_1 \rightarrow \mathcal{I}$, $F_2 \wedge \mathcal{I}$ is unsatisfiable, and $\mathcal{I}$ refers only to the intersection of variables in F_1 and F_2.



(a) A losing AGT T



(b) Tree slice T_1



(c) Tree slice T_2

Figure 6.2: Splitting of an abstract game tree by the learning procedure.

We choose a non-leaf node n of T with maximal depth, i.e., a node whose children are leafs (Algorithm 13, line 3). We then split the tree at n such that both slices T_1 and T_2 contain a copy of n (line 4). Figure 6.2b shows T_1, which contains all of T except n's children, and T_2 (Figure 6.2c), which contains only n and its children. There is no candidate strategy for T so $s \wedge \overline{\text{TREEFORMULA}}(k, T)$ is unsatisfiable. By construction, $(\overline{\text{TREEFORMULA}}(k, T_1) \wedge \overline{\text{TREEFORMULA}}(k, T_2)) \implies \overline{\text{TREEFORMULA}}(k, T)$ and so we know that $(s \wedge \overline{\text{TREEFORMULA}}(k, T_1) \wedge \overline{\text{TREEFORMULA}}(k, T_2))$ is also unsatisfiable.

We construct an interpolant with $F_1 = s(\mathcal{S}_T) \wedge \text{TREEFORMULA}(k, T_1)$ and $F_2 = \text{TREEFORMULA}(k, T_2)$ (line 5). The only variables shared between F_1 and F_2 are the state variable copies belonging to node n. By the properties of the interpolant, $F_2 \wedge \mathcal{I}$ is unsatisfiable, therefore all states in $\mathcal{I}$ are losing against abstract game tree T_2 in Figure 6.2c. We also know that $F_1 \rightarrow \mathcal{I}$, thus $\mathcal{I}$ contains all states reachable at n by following T_1 and avoiding error states.

At node n, the interpolant $(\texttt{nrequests} = 1 \wedge \texttt{resource1} = 1)$ captures the information we need. Any action by the environment followed by one of the controller actions at n will be winning for the controller.

We have discovered a set $\mathcal{I}$ of states losing for the environment. Environment-

Algorithm 13 Learning algorithms

Require: $s(\mathcal{S}_T) \wedge \textsc{treeFormula}(k, T) \equiv \bot$
Require: *Must-invariant* holds
Ensure: *Must-invariant* holds
Ensure: $s(\mathcal{S}_T) \wedge B^M \not\equiv \bot$ ▷ s will be added to B^M
1: **function** $\textsc{learn}(s, T)$
2: **if** $\textsc{succ}(T) = \emptyset$ **then return**
3: $n \leftarrow$ non-leaf node with min height
4: $\langle T_1, T_2 \rangle \leftarrow \textsc{gtSplit}(T, n)$
5: $F_1 \leftarrow s(\mathcal{S}_T) \wedge \textsc{treeFormula}(k, T_1)$
6: $F_2 \leftarrow \textsc{treeFormula}(k, T_2)$
7: $\mathcal{I} \leftarrow \textsc{interpolate}(F_1, F_2)$
8: $B^M \leftarrow B^M \vee \mathcal{I}$
9: $\textsc{learn}(s, T_1)$
10: **end function**
Require: $s(\mathcal{S}_T) \wedge \overline{\textsc{treeFormula}}(k, T) \equiv \bot$
Require: *May-invariant* holds
Ensure: *May-invariant* holds
Ensure: $s(\mathcal{S}_T) \wedge B^m[\textsc{height}(k, T)] \equiv \bot$ ▷ s will be removed from B^m
11: **function** $\overline{\textsc{learn}}(s, T)$
12: **if** $\textsc{succ}(T) = \emptyset$ **then return**
13: $n \leftarrow$ non-leaf node with min height
14: $\langle T_1, T_2 \rangle \leftarrow \textsc{gtSplit}(T, n)$
15: $F_1 \leftarrow s(\mathcal{S}_T) \wedge \overline{\textsc{treeFormula}}(k, T_1)$
16: $F_2 \leftarrow \overline{\textsc{treeFormula}}(k, T_2)$
17: $\mathcal{I} \leftarrow \textsc{interpolate}(F_1, F_2)$
18: **for** $i = 1$ to $\textsc{height}(k, n)$ **do**
19: $B^m[i] \leftarrow B^m[i] \setminus \mathcal{I}$
20: **end for**
21: $\overline{\textsc{learn}}(s, T_1)$
22: **end function**

losing states are only losing for a particular bound: given that there does not exist an environment strategy that forces the game into an error state in k rounds or less; there may still exist a longer environment-winning strategy. We therefore record learned environment-losing states along with associated bounds. To this end, we maintain a conceptually infinite array of sets $B^m[k]$ that are may-losing for the controller, indexed by bound k. $B^m[k]$ are initialised to E for all k. Whenever an environment-losing set $\mathcal{I}$ is discovered for a node n with bound $\textsc{height}(k, n)$ in line 13 of Algorithm 13, this set is subtracted from $B^m[i]$, for all i less than or equal to the bound (lines 14–16).

Algorithm 14 Amended tree formulas for Controller and Environment

1: **function** TREEFORMULA(k, T)
2: **if** HEIGHT$(k, T) = 0$ **then**
3: **return** $\neg B^M(\mathcal{S}_T)$
4: **else**
5: **return** $\neg B^M(\mathcal{S}_T) \wedge$
6:

$$\bigwedge_{\langle e,n\rangle \in \text{SUCC}(T)} (\delta(\mathcal{S}_T, U_e, C_n, \mathcal{S}_n) \wedge U_e = \text{ACTION}(e) \wedge \text{TREEFORMULA}(k, n))$$

7: **end if**
8: **end function**
9: **function** $\overline{\text{TREEFORMULA}}(k, T)$
10: **if** HEIGHT$(k, T) = 0$ **then**
11: **return** E$(\mathcal{S}_T)$
12: **else**
13: **return** $B^m[\text{HEIGHT}(k, T)](\mathcal{S}_T) \wedge$
14:

$$\left(\text{E}(\mathcal{S}_T) \vee \bigvee_{\langle e,n\rangle \in \text{SUCC}(T)} (\delta(\mathcal{S}_T, U_n, C_e, \mathcal{S}_n) \wedge C_e = \text{ACTION}(e) \wedge \overline{\text{TREEFORMULA}}(k, n))\right)$$

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

The $\overline{\text{TREEFORMULA}}$ function is modified for the unbounded solver (Algorithm 14) to constrain the environment to the appropriate B^m. This enables further interpolants to be constructed by the learning procedure recursively splitting more nodes from T_1 (Algorithm 13, line 7) since the states that are losing to T_2 are no longer contained in B^m.

Learning of states losing for the controller is similar (LEARN in Algorithm 13). The main difference is that environment-losing states are losing for all bounds. Therefore we record these states in a single set B^M of must-losing states (Algorithm 13, line 6). This set is initialised to the error set E and grows as new losing states are discovered. The modified $\overline{\text{TREEFORMULA}}$ function (Algorithm 14) blocks must-losing states, which also allows for recursive learning over the entire tree.

6.1.2 Main synthesis loop

Figure 15 shows the main loop of the unbounded synthesis algorithm. The algorithm invokes the modified bounded synthesis procedure with increasing bound k until the initial state is in B^M (environment wins) or B^m reaches a fixed point (controller wins). We prove correctness in the next section.

Algorithm 15 Unbounded Synthesis

```
1:  function SOLVEUNBOUNDED(T)
2:      B^M ← E
3:      B^m[0] ← E
4:      for k = 1 ... do
5:          if SAT(s_0 ∧ B^M) then
6:              ▷ Losing in the initial state
7:              return unrealisable
8:          end if
9:          if ∃i < k. B^m[i] ≡ B^m[i + 1] then
10:             ▷ Reached fixed point
11:             return realisable
12:         end if
13:         B^m[k] ← E
14:         CHECKBOUND(k)
15:     end for
16: end function
```

Require: *May* and *must* invariants hold
Ensure: *May* and *must* invariants hold
Ensure: $s_0 \notin B^m[k]$ if there exists a winning controller strategy with bound k
Ensure: $s_0 \in B^M$ if there exists a winning environment strategy with bound k

```
17: function CHECKBOUND(k)
18:     return SOLVEABSTRACT(env, s_0, k, ∅)
19: end function
```

6.2 Correctness

We define two global invariants of the algorithm. The *may-invariant* states that sets $B^m[i]$ grow monotonically with i and that each $B^m[i+1]$ overapproximates the states from which the environment can force the game into $B^m[i]$. We call this operation $Upre$, the uncontrollable predecessor. So the

may-invariant is:

$$\forall i < k.\ B^m[i] \subseteq B^m[i+1], Upre(B^m[i]) \subseteq B^m[i+1].$$

The *must-invariant* guarantees that the must-losing set B^M is an underapproximation of the actual losing set B:

$$B^M \subseteq B.$$

Correctness of SOLVEUNBOUNDED follows from these invariants. The must-invariant guarantees that the environment can force the game into an error state from B^M, therefore checking whether the initial state is in B^M (as in line 5) is sufficient to return `unrealisable`. The may-invariant tells us that if $B^m[i] \equiv B^m[i+1]$ (line 6) then $Upre(B^m[i]) \subseteq B^m[i]$, i.e. $B^m[i]$ overapproximates the winning states for the environment. We know that $s_0 \notin B^m[k]$ due to the post-condition of CHECKBOUND, and since the may-invariant tells us that B^m is monotonic then s_0 must not be in $B^m[i]$. If $s_0 \notin B^m[i]$ then s_0 is not in the winning states for the environment and the controller can always win from s_0.

Both invariants trivially hold after B^m and B^M have been initialised in the beginning of the algorithm. The sets B^m and B^M are only modified by the functions LEARN and $\overline{\text{LEARN}}$. Below we prove that $\overline{\text{LEARN}}$ maintains the invariants. The proof of LEARN is similar.

6.2.1 Proof of $\overline{\text{learn}}$

We prove that postconditions of $\overline{\text{LEARN}}$ are satisfied assuming that its preconditions hold.

Line (11–12) splits the tree T into T_1 and T_2, such that T_2 has depth 1. Consider formulas $F_1 = s(\mathcal{S}_T) \wedge \overline{\text{TREEFORMULA}}(k, T_1)$ and $F_2 = \overline{\text{TREEFORMULA}}(k, T_2)$. These formulas only share variables $\mathcal{S}_n$. Their conjunction $F_1 \wedge F_2$ is unsatisfiable, as by construction any solution of $F_1 \wedge F_2$ also satisfies $s(\mathcal{S}_T) \wedge \overline{\text{TREEFORMULA}}(k, T)$, which is unsatisfiable (precondition (b)). Hence the interpolation operation is defined for F_1 and F_2.

Intuitively, the interpolant computed in line (13) overapproximates the set of states reachable from s by following the tree from the root node to n, and underapproximates the set of states from which the environment loses against tree T_2.

Formally, $\mathcal{I}$ has the property $\mathcal{I} \wedge F_2 \equiv \bot$. Since T_2 is of depth 1, this means that the environment cannot force the game into $B^m[\text{HEIGHT}(k,n)-1]$ playing against the counterexample moves in T_2. Hence, $\mathcal{I} \cap Upre(B^m[\text{HEIGHT}(k,n)-1]) = \emptyset$. Furthermore, since the may-invariant holds, $\mathcal{I} \cap Upre(B^m[i]) = \emptyset$, for all $i < \text{HEIGHT}(k,n)$. Hence, removing $\mathcal{I}$ from all $B^m[i], i \leq \text{HEIGHT}(k,n)$ in line (15) preserves the may-invariant, thus satisfying the first post-condition.

Furthermore, the interpolant satisfies $F_1 \rightarrow \mathcal{I}$, i.e., any assignment to $\mathcal{S}_n$ that satisfies $s(\mathcal{S}_T) \wedge \overline{\text{TREEFORMULA}}(k, T_1)$ also satisfies $\mathcal{I}$. Hence, removing $\mathcal{I}$ from $B^m[\text{HEIGHT}(k,n)]$ makes $s(\mathcal{S}_T) \wedge \overline{\text{TREEFORMULA}}(k, T_1)$ unsatisfiable, and hence all preconditions of the recursive invocation of $\overline{\text{LEARN}}$ in line (17) are satisfied.

At the second last recursive call to $\overline{\text{LEARN}}$, tree T_1 is empty, n is the root node, $\overline{\text{TREEFORMULA}}(k, T_1) \equiv B^m[\text{HEIGHT}(k, T_1)](\mathcal{S}^T)$; hence $s(\mathcal{S}_T) \wedge \overline{\text{TREEFORMULA}}(k, T_1) \equiv s(\mathcal{S}_T) \wedge B^m[\text{HEIGHT}(k, T_1)](\mathcal{S}^T) \equiv \bot$. Thus the second postcondition of $\overline{\text{LEARN}}$ holds.

The proof of LEARN is similar to the above proof of LEARN. An interpolant constructed from $F_1 = s(\mathcal{S}_T) \wedge \text{TREEFORMULA}(k, T_1)$ and $F_2 = \text{TREEFORMULA}(k, T_2)$ has the property $\mathcal{I} \wedge F_2 \equiv \bot$ and the precondition ensures that the controller is unable to force the game into B^M playing against the counterexample moves in T_2. Thus adding $\mathcal{I}$ to B^M maintains the must-invariant satisfying the first postcondition.

Likewise, in the second last recursive call of LEARN with the empty tree T_1 and root node n: $\text{TREEFORMULA}(k, T_1) \equiv \neg B^M(\mathcal{S}_T)$. Hence $s(\mathcal{S}_T) \wedge \text{TREEFORMULA}(k, T_1) \equiv s(\mathcal{S}_T) \wedge \neg B^M(\mathcal{S}_T) \equiv \bot$. Therefore $s(\mathcal{S}_T) \wedge B^M(\mathcal{S}_T) \not\equiv \bot$, the second postcondition, is true.

6.2.2 Proof of Termination

We must prove that CHECKBOUND terminates and that upon termination its postcondition holds, i.e., state s_0 is removed from $B^m[\kappa]$ if there is a winning controller strategy on the bounded safety game of maximum bound κ or it is added to B^M otherwise. Termination follows from completeness of counterexample guided search, which terminates after enumerating all possible opponent moves in the worst case.

Assume that there is a winning strategy for the controller at bound κ. This means that at some point the algorithm discovers a counterexample tree of bound κ for which the environment cannot force into E. The algorithm then

invokes the $\overline{\text{LEARN}}$ method, which removes s_0 from $B^m[\kappa]$. Alternatively, if there is a winning strategy for the environment at bound κ then a counterexample losing for the controller will be found. Subsequently LEARN will be called and s_0 added to B^M.

6.3 Optimisations

6.3.1 Generalising the initial state

This optimisation allows us to learn may and must losing states faster. Starting with a larger set of initial states we increase the reachable set and hence increase the number of states learned by interpolation. This optimisation requires a modification to SOLVEABSTRACT to handle sets of states, which is not shown.

The optimisation is relatively simple and is inspired by a common greedy heuristic for minimising `unsat` cores. Initial state s_0 assigns a value to each variable in $\mathcal{S}$. If the environment loses $\langle s_0, k\rangle$ then we attempt to solve for a generalised version of s_0 by removing one variable assignment at a time. If the environment loses from the larger set of states then we continue generalising. In this way we learn more states by increasing the reachable set. In our benchmarks we have observed that this optimisation is beneficial on the first few iterations of CHECKBOUND.

Algorithm 16 Generalise s_0 optimisation

```
function CHECKBOUND(k)
    r ← SOLVEABSTRACT(env, s_0, k, ∅)
    if r ≠ ∅ then return r
    s' ← s_0
    for x ∈ S do
        r ← SOLVEABSTRACT(env, s' \ {x}, k, ∅)
        if r = NULL then
            ▷ Remove the assignment to x from s'
            s' ← s' \ {x}
        end if
    end for
    return NULL
end function
```

6.3.2 Generalising losing states

The same generalisation operation can be performed during computational learning. When learning with interpolation the size of the learned set can vary from exactly the reachable states to a large overapproximation of reachable states. The interpolant is constructed with the property $F_1 \implies \mathcal{I}$, so by increasing the number of states represented by F_1 we also increase the size of $\mathcal{I}$. F_1 is given by $s(\mathcal{S}_{T_1}) \wedge \overline{\text{TREEFORMULA}}(T_1, k)$ and represents the set of states reachable from s by playing the actions in T_1. For correctness we require that s is losing to the full tree T so we can increase the size of s by generalising in the same way as the previous optimisation. We drop variables from s such that the new cube s' is also losing to the actions in T. This increases F_1 and we may learn a larger interpolant.

6.3.3 Improving candidate strategies

In Chapter 4 I introduced strategy shortening to improve candidate strategies. For unbounded synthesis this optimisations is even more useful because it allows the algorithm to learn losing states more quickly. Strategy shortening attempts to push *good* actions closer to the root of the game tree. For learning environment-losing states this can mean that states can be proven to be losing for larger values of k.

As long as FINDCANDIDATE returns a strategy whenever one exists there is significant freedom for heuristics to choose strategies without affecting correctness. For example, an additional SAT call in FINDCANDIDATE can first check for the existence of a strategy that loops by adding the requirement that $\exists i \exists j (s_i = s_j)$ in all runs $s_0, ..., s_k$. The motivation behind this optimisation was that a strategy that forces a loop will quickly converge on a fixed point. In practice, such strategies are rarely found and the additional call to the SAT solver was a waste of resources. It remains future work to fully explore the possibilities of heuristically selecting candidates. One avenue for exploration is the use of QBF or 2QBF solvers to check for actions that are winning for all opponent actions, i.e. partially reintroducing universal quantifiers to the search.

6.4 Discussion

Unbounded realisability is designed to take advantage of the strengths of bounded realisability but provide the completeness offered by a fixed point computation. These conflicting aims are addressed by overapproximation via interpolation, which allows completeness without sacrificing performance to an exploding symbolic representation.

6.4.1 Related work

Synthesis of safety games is a thoroughly explored area of research with most efforts directed toward solving games with BDDs [Burch et al., 1990] and abstract interpretation [Brenguier et al., 2014; Walker and Ryzhyk, 2014]. Satisfiability solving has been used previously for synthesis in a suite of methods proposed by Bloem et al. [Bloem et al., 2014]. The authors propose employing competing SAT solvers to learn clauses representing bad states, which is similar to our approach but does not unroll the game. They also suggest QBF solver, template-based, and Effectively Propositional Logic (EPR) approaches.

SAT-based bounded model checking approaches that unroll the transition relation have been extended to unbounded by using conflicts in the solver [McMillan, 2002], or by interpolation [McMillan, 2003]. However, there are no corresponding adaptations to synthesis.

Incremental induction [Bradley, 2011] is another technique for unbounded model checking that inspired several approaches to synthesis including the work presented here. Morgenstern et al. [Morgenstern et al., 2013] proposed an technique that computes sets of states that overapproximate the losing states (similar to our B^m) and another set of winning states (similar to the negation of B^M). Their algorithm maintains a similar invariant over the sets of losing states as our approach and has the same termination condition. It differs in how the sets are computed, which it does by inductively proving the number of game rounds required by the environment to win from a state. Chiang and Jiang [Chiang and Jiang, 2015] recently proposed a similar approach that focusses on computing the winning region for the controller forwards from the initial state in order to take advantage of reachability information and bad transition learning without needing to discard learnt clauses.

There are approaches to synthesis of LTL specifications that use bounds to simplify the problem. The authors of [Finkbeiner and Schewe, 2013] suggest a methodology directly inspired by bounded model checking and it has been adapted to symbolic synthesis [Ehlers, 2012]. In contrast to the approach here the bound is placed on the implementation instead of the number of game rounds. Lazy synthesis [Finkbeiner and Jacobs, 2012] similarly constructs implementations of a bounded size but does so in a counterexample guided approach. Their approach is to use an SMT solver to produce a candidate implementation and then check the implementation with a BDD based model checker. These bounded synthesis techniques are similar in idealogy to the approach described here but are used to solve a different problem.

6.4.2 Limitations

Bounded synthesis is generally efficient for games without a high branching factor, as discussed in Chapter 4. This limitation affects the unbounded solver as can be seen in the synthesis competition results on specifications such as the adder. In a correct adder the controller must set a variable c to be equal to $a + b$ where a and b are environment variables. The unbounded synthesis algorithm must construct a game tree consisting of all possible values to c in order to prove realisability.

The unbounded solver extends the bounded solver with learning. Learning states from the game tree does not introduce significant complexity to the problem. However, there are cases in which learning can be slow to converge on a fixed point. As a result the bounded algorithm must be iterated many times with increasing bounds.

In the synthesis competition benchmarks there is a simple counter specification that helps illustrate the limitations of unbounded synthesis. The specification is given a parameter n that determines the number of bits in the counter. Figure 6.3 shows the specification. The environment has the choice to increment the counter, or not. The controller can reset the counter when it is half way to the maximum value. The controller is safe if the counter never reaches its maximum value. Clearly a safe controller resets whenever it is able to.

This example can produce several different interesting behaviours in the unbounded synthesis algorithm. Let us first consider the game tree in Figure 6.4a. The controller strategy shown in this tree is to not reset the counter.

Uncontrollable	Controllable	State
stay = {0,1}	reset = {0,1}	counter = {0,..,2^n-1}
		err = {0,1}

(a) Variables

```
counter = 0; err = 0;
```

(b) Initial State

```
counter' = if (stay)
             counter
           else if (counter == (2^n-1)-1 ∧ reset)
             0
           else
             counter + 1
    err' = (counter == 2^n-1)
```

(c) Transition Relation

Figure 6.3: Parameterised counter example

This strategy wins the bounded game for any counter with $n \geq 2$ because there are not enough game rounds for the counter to reach the error state. Let's now consider how the unbounded algorithm would learn from this tree when $n = 3$. At the node of height 1, the largest interpolant that could be learned is $((\texttt{err == 0}) \wedge (\texttt{counter < 6}))$. At the next node up the tree we might learn $((\texttt{err == 0}) \wedge (\texttt{counter < 5}))$. With a tree of length $k = 2^n - 2 = 6$ we could have B^m with a difference of exactly one state in each successive index. For high values of n it is infeasible to construct a formula in CNF for this many unrollings of the transition relation, even though there is no branching.

In practise, unbounded synthesis can do well on this specification when the SAT solver gives a candidate such as in Figure 6.4b. The optimisation that generalises the losing state before interpolation is able to drop all bits of the counter except the high bit. This gives $((\texttt{err = 0}) \wedge (\texttt{counter < 3}))$, which describes the safe region of the game. The procedure quickly converges on this fixed point in this case.

(a) No reset candidate (b) Reset candidate

Figure 6.4: Game trees for the counter specification

6.4.3 Strengths

The strength of unbounded synthesis lies in the ability to approximate the winning region. This can be seen on a scaled up version of the example used in Section 6.1. A system with n resources and an environment that can make m requests at once is always realisable if $n \geq m$ by a controller that always grants access to m resources. The problem for BDD solvers is that it can be difficult to compactly represent all ${}^{n}C_{m}$ combinations of granted resources that make up the winning region. With unbounded synthesis a fixed point can be reached by choosing just one winning combination and proving that the environment cannot force an escape from that small set of states.

In Chapter 4 I showed that the strength of the bounded realisability algorithm lies in quickly discovering counterexamples that are difficult to find by representing the entire winning or losing region as a BDD. Unbounded synthesis has similar advantages except that we can extend this advantage to specifications that are realisable.

6.5 Summary

I have now shown how to extend bounded realisability to unbounded games by interpolating abstract game trees to learn an overapproximation of the environment's winning region. The resulting algorithm is a sound and complete procedure for realisability that is efficient is certain cases where BDD based methods are not. In the next chapter I will present an implementation of the algorithm and show results.

- Chapter 4 introduced bounded realisability, which constructs game trees

as abstractions of the game. An optimisation was described that prunes the search tree by learning the set of states that are losing for a particular abstraction. In this chapter states are learned from the same game trees with interpolation.

- Learning with interpolants ensures that certain properties are maintained on the losing states. By carefully maintaining invariants using those properties a fixed point in environment losing states can be detected.
- The constructed fixed point is an overapproximation of the total set of environment losing states. Unbounded synthesis can be more efficient than BDD solvers in cases where computing the entire set of states is costly.

7 Evaluation

7.1 Bounded realisability

We evaluate our algorithm on four families of benchmarks derived from driver synthesis problems. These benchmarks model the data path of four I/O devices in the abstracted form. In particular, we model the transmit buffer of an Ethernet adapter, the send queue of a UART serial controller, the command queue of an SPI Flash controller, and the IDE hard disk DMA descriptor list. Models are parameterised by the size of the corresponding data structure. Specifications are written in a simple input language based on the NuSMV syntax [**?**]. The transition relation of the game is given in the form of variable update functions $x := f(X, Y_c, Y_u)$, one for each state variable $x \in X$.

We compare our solver against two existing approaches to solving games. First, we encode input specifications as QBF instances and solve them using two state-of-the-art QBF solvers: RAReQS [**?**] and depqbf [**?**], having first run them through the bloqqer [**?**] preprocessor. Second, we solve our benchmarks using the Termite [**?**] BDD-based solver that uses dynamic variable reordering, variable grouping, transition relation partitioning, and other optimisations.

Our experiments, summarised in Figure 7.5, show that off-the-shelf QBF solvers are not well-suited for solving games. Although our algorithm is inspired by RAReQS, we achieve much better performance, since our solver takes into account the structure of the game, rather than treating it as a generic QBF problem.

All four benchmarks have very large sets of winning states. Nevertheless, in the UART and IDE benchmarks, Termite is able to represent winning states compactly with only a few thousand BDD nodes. It scales well and

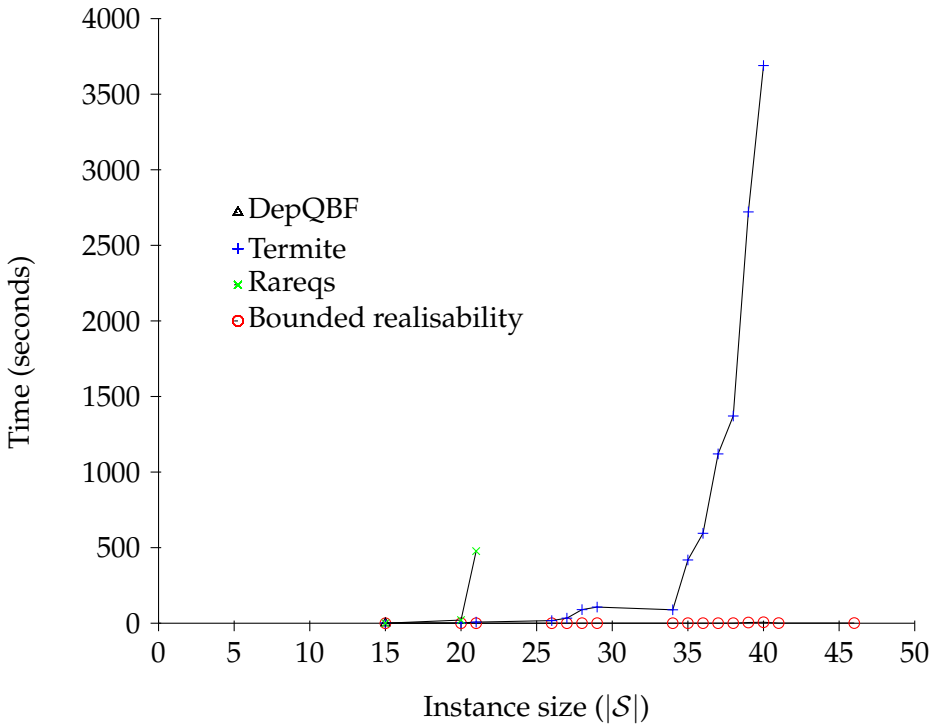


Figure 7.1: UART

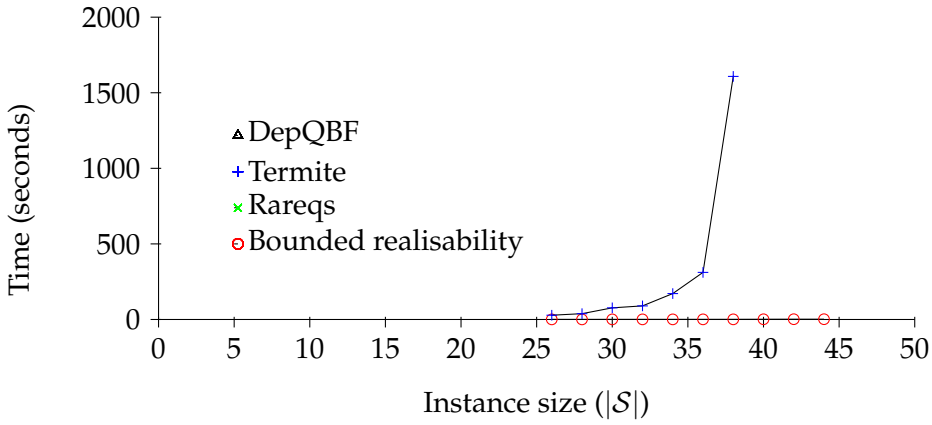


Figure 7.2: IDE DMA

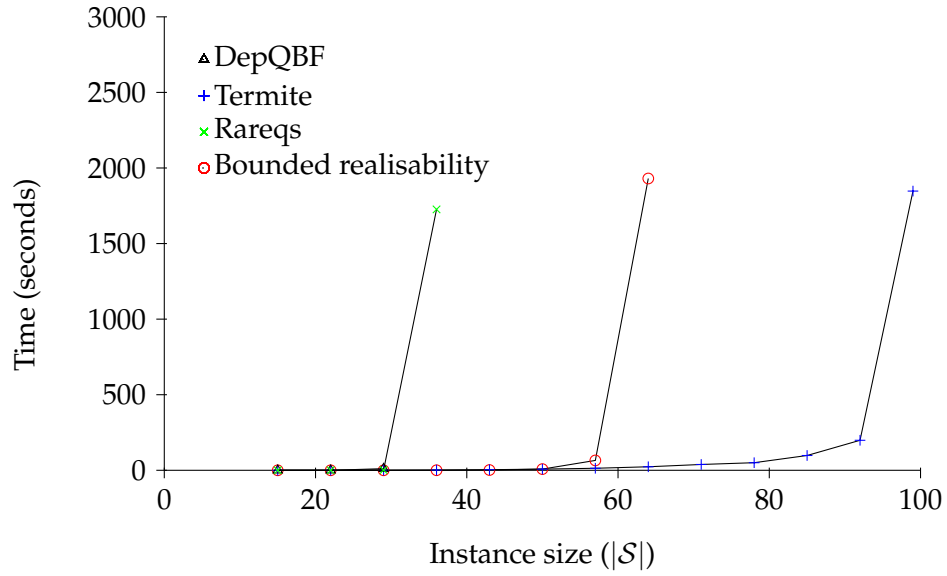


Figure 7.3: SPI

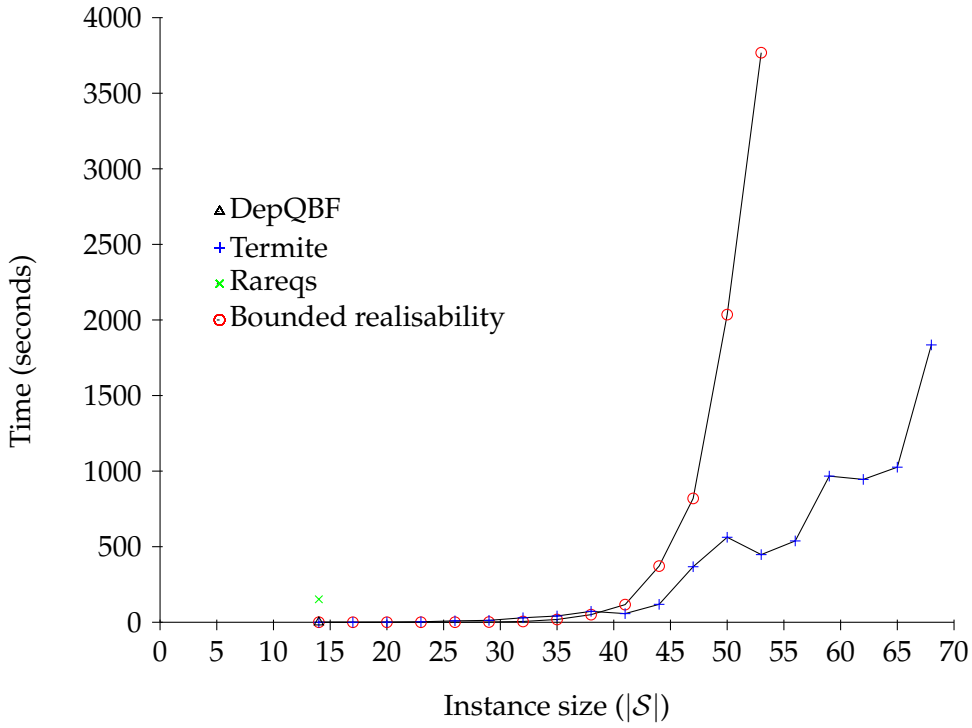


Figure 7.4: queue

outperforms EvaSolveron these benchmarks. However, in the two other benchmarks, Termite does not find a compact BDD-based representation of the winning set. EvaSolveroutperforms Termite on these benchmarks as it does not try to enumerate all winning states.

Detailed performance analysis shows that abstract game trees generated in our benchmarks had average branching factors in the range between 1.03 and 1.2, with the maximal depth of the trees ranging from 3 to 58. This confirms the the key premise behind the design of EvaSolver, namely, solving real-world synthesis problems requires considering only a small number of opponent moves in every state of the game.

7.2 Strategy extraction

We implemented our strategy generation algorithm as an extension on top of $EvaSolver$. We use PeRIPLO [**?**] to generate interpolants in Algorithms 9–10. We use BDDs to compile and store winning sets W in Algorithm 9 [**?**]. $EvaSolver$ implements an important optimisation whereby it generates multiple certificate trees for fragments of the original game, which enables computational learning of winning states. Our strategy generation algorithm is invoked in an online fashion, whenever $EvaSolver$ computes a certificate tree for a fragment. This results in multiple partial strategies, where each strategy is computed as described in the previous section. We introduce an additional final step to $EvaSolver$, which combines partial strategies into a global winning strategy for the original game.

We evaluate our implementation on driver synthesis benchmarks from [**?**]. These benchmarks model the data path of four I/O devices in the abstracted form. Each specification was parameterised to scale in the size of the state space and the winning strategy length. The four families are UART, SPI, IDE and Ethernet, which model respectively the send queue of a UART serial device, the command queue of an SPI device, the DMA descriptor list of an IDE controller, and the transmit buffer of an Ethernet device respectively. On two of these families, namely SPI and Ethernet, $EvaSolver$ outperforms a state-of-the-art BDD-based game solver.

Figure 7.5 summarises our results. For each family, we show strategy generation time as a function of the number of state variables in the specification for 5 hardest instances of the family solved by $EvaSolver$. Specifically, we

(a) IDE (b) UART

(c) SPI Flash (d) Ethernet

Figure 7.5: Performance of strategy extraction algorithm on four parameterised benchmarks. The X-axis shows the number of state vars in the game.

show the time it took the base solver to determine the winner (the WD line), as well as the total time taken to solve the game and compute the winning strategy using our algorithm (WD+Strategy).

Profiling showed that non-negligible overhead was introduced by transferring CNFs from $EvaSolver$'s internal representation to the representation used by the interpolation library. This overhead can be almost completely eliminated with additional engineering effort. Hence, we also show the effective time spent solving the game and computing the strategy, excluding the formula translation time (the Effective Time line).

Table 7.1 shows a detailed breakdown of experimental results, including the number of state variables for each instance (**Vars**) and the total time taken by the solver (**Total(s)**), split between the time used to determine the winner (and generate certificate trees) (**Base(s)**) and the strategy generation time (**Strategy(s)**). The **OH** and **EffOH** columns show total and effective overheads introduced by the strategy generation step.

The **Size** column shows the size of the strategy, i.e., the number of (W, a, k) tuples returned by the GENLOCALSTRATS function. The last three columns report on our use of the PeRIPLO interpolation library in terms of the number of interpolation operations performed by the algorithm when solving the instance, the average and the maximal size of interpolants returned by PeRIPLO. Numbers in brackets show the size of the interpolant compiled to a BDD.

Our results show that strategy generation adds a modest overhead to the base solver. Effective overheads are about 12% for IDE and SPI, about 35% for Ethernet and about 30% for UART. Most of this overhead is due to interpolant computation. Moreover, experiments show that our algorithm scales linearly with the time taken by the base solver to determine the winner.

These results show that our algorithm is efficient, scalable and robust. The last property is particularly interesting, as existing strategy extraction algorithms for traditional game solvers, based on winning set compilation, have been reported to introduce significant variance across instances [**??**]. A conclusive comparison can only be performed by evaluating both types of

algorithms on a common set of benchmarks. Such a comparison requires first extending $EvaSolver$ to support unbounded safety and reachability games and is part of the future work.

7.3 Unbounded realisability

We evaluate our approach on the benchmarks of the 2015 synthesis competition (SYNTCOMP'15). Each benchmark comprises of controllable and uncontrollable inputs to a circuit that assigns values to latches. One latch is configured as the error bit that determines the winner of the safety game. The benchmark suite is a collection of both real-world and toy specifications including generalised buffers, AMBA bus controllers, device drivers, and converted LTL formulas. Descriptions of many of the benchmark families used can be found in the 2014 competition report [**?**].

The implementation of our algorithm uses GLUCOSE [**?**] for SAT solving and PERIPLO [**?**] for interpolant generation. We intend to open source the tool for SYNTCOMP'16. The benchmarks were run on a cluster of Intel Quad Core Xeon E5405 2GHz CPUs with 16GB of memory. The solvers were allowed exclusive access to a node for one hour to solve an instance.

The results of our benchmarking are shown, along with the synthesis competition results [**?**], in Table 7.2. The competition was run on Intel Quad Core 3.2Ghz CPUs with 32GB of memory, also on isolated nodes for one hour per instance. The competition results differ significantly from our own benchmarks due to this more powerful hardware. For our benchmarks we report only the results for solvers we were able to run on our cluster. The unique column lists the number of instances that only that tool could solve in the competition (excluding our solver). In brackets is the number of instances that only that tool could solve, including our solver.

Our implementation was able to solve 103 out of the 250 specification in the alloted time, including 12 instances that were not solved by any other solver in the sequential track of the competition. The unique instances we solved are listed in Table 7.3.

Five of the instances unique to our solver are device driver instances and another five are from the `stay` family. This supports the hypothesis that different game solving methodologies perform better on certain classes of specifications.

We also present a cactus plot of the number of instances solved over time (Figure 7.6). We have plotted the best configuration of each solver we benchmarked. The solvers shown are DEMIURGE [**?**], the only SAT-based tool in the competition, the winner of the sequential realisability track SIMPLE BDD SOLVER 2 [**?**], and AbsSynthe (seq3) [**?**].

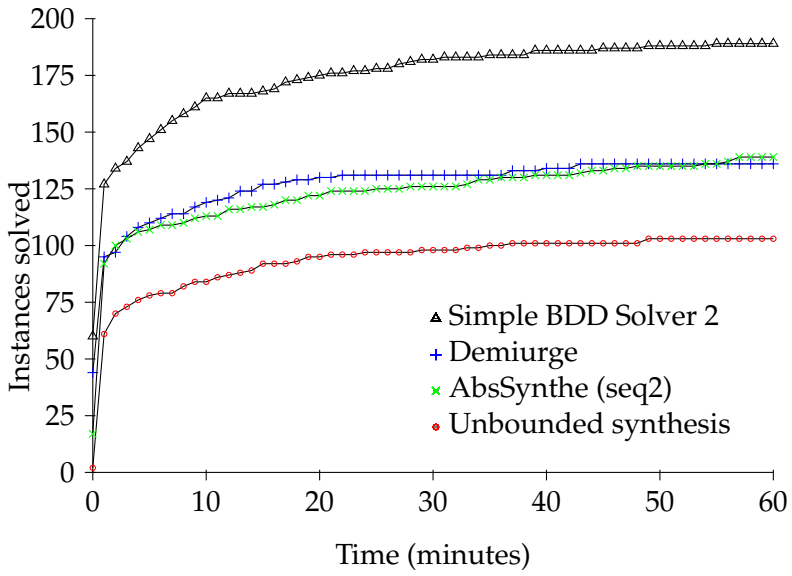


Figure 7.6: Number of instances solved over time.

Whilst our solver is unable to solve as many instances as other tools, it was able to solve more unique instances than any solver in the competition. This confirms that our methodology is able to fill gaps in a state of the art synthesis toolbox by more efficiently solving instances that are hard for other techniques. For this reason our solver would be a worthwhile addition to a portfolio solver. In the parallel track of the competition, DEMIURGE uses a suite of 3 separate but communicating solvers. The solvers relay unsafe states to one another, which is compatible with the set B^M in our solver. This technique can already solve each of the unique instances solved by our solver but there may still be value in the addition of this work to the portfolio. It remains future work to explore this possibility.

Vars	Total(s)	Base(s)	Strategy(s)	OH	EffOH	Size	INum	IAvg	IMax
IDE benchmark									
26	32.62	25.42	7.20	1.28	1.16	50	48	1193 (23)	24155 (118)
28	42.20	35.49	6.72	1.19	1.10	59	52	2489 (27)	39384 (119)
30	60.04	51.93	8.11	1.16	1.08	92	43	72 (17)	1583 (148)
32	115.11	107.35	7.77	1.07	1.04	60	36	18 (14)	85 (27)
34	283.08	227.67	55.40	1.24	1.21	159	49	2150 (15)	103941 (38)
SPI benchmark									
15	0.35	0.26	0.09	1.36	1.26	8	5	11 (9)	21 (9)
22	0.94	0.72	0.22	1.31	1.19	15	12	10 (10)	22 (18)
29	2.46	1.90	0.56	1.29	1.16	22	17	12 (10)	25 (18)
36	3.56	2.91	0.65	1.22	1.11	107	22	11 (10)	22 (18)
43	9.11	7.09	2.03	1.29	1.13	166	27	11 (10)	21 (14)
50	16.20	12.85	3.35	1.26	1.12	233	32	11 (11)	23 (18)
57	25.00	19.86	5.14	1.26	1.12	322	37	12 (11)	21 (18)
64	38.48	31.48	7.00	1.22	1.10	416	42	12 (11)	24 (18)
71	57.88	47.94	9.94	1.21	1.09	70	47	12 (12)	21 (18)
78	91.51	75.02	16.49	1.22	1.10	636	52	12 (12)	22 (19)
85	141.10	116.71	24.39	1.21	1.09	773	57	13 (12)	23 (20)
92	193.96	162.05	31.91	1.20	1.09	917	62	13 (13)	24 (21)
99	309.44	256.88	52.55	1.20	1.09	1059	67	13 (13)	22 (22)
106	449.49	377.48	72.00	1.19	1.09	1223	72	13 (13)	23 (23)
113	1645.44	1543.84	101.60	1.07	1.03	117	77	14 (13)	24 (24)
120	901.95	830.17	71.77	1.09	1.04	1637	82	14 (14)	25 (25)
127	2259.65	2143.40	116.25	1.05	1.02	139	87	14 (14)	26 (26)
134	2385.74	2193.65	192.09	1.09	1.04	152	92	14 (14)	27 (27)
Ethernet benchmarks									
14	0.06	0.03	0.02	1.60	1.52	2	1	24 (13)	24 (13)
17	0.49	0.29	0.20	1.70	1.45	21	7	33 (16)	87 (30)
20	1.97	1.14	0.82	1.72	1.45	176	15	41 (16)	110 (26)
23	5.39	3.23	2.16	1.67	1.39	185	25	64 (23)	180 (42)
26	14.61	7.94	6.66	1.84	1.48	266	36	100 (24)	347 (45)
29	27.41	15.71	11.70	1.74	1.43	677	44	155 (24)	779 (48)
32	58.02	35.38	22.64	1.64	1.36	208	61	136 (28)	676 (55)
35	111.69	69.26	42.43	1.61	1.35	351	80	141 (31)	933 (75)
38	238.09	151.21	86.89	1.57	1.33	545	116	184 (32)	1081 (63)
41	513.61	321.78	191.82	1.60	1.34	1525	154	184 (35)	1123 (72)
44	845.51	530.68	314.83	1.59	1.34	2159	191	276 (37)	2253 (64)
47	903.79	590.19	313.60	1.53	1.30	1547	228	261 (38)	1780 (71)
50	1368.23	875.90	492.33	1.56	1.33	1670	236	372 (38)	2292 (85)
UART benchmarks									
15	2.86	2.19	0.67	1.31	1.19	12	40	28 (6)	90 (14)
20	3.16	2.33	0.83	1.36	1.20	20	14	35 (12)	155 (23)
21	10.06	6.96	3.09	1.44	1.24	35	34	48 (9)	306 (26)
26	27.89	18.55	9.34	1.50	1.27	65	60	92 (13)	730 (41)
27	63.68	41.49	22.20	1.53	1.29	93	94	96 (13)	825 (44)
28	137.24	90.68	46.56	1.51	1.27	103	136	138 (13)	1356 (42)
29	270.66	178.75	91.92	1.51	1.27	134	184	212 (15)	2806 (47)
34	553.29	360.76	192.53	1.53	1.28	191	246	299 (16)	6360 (54)
35	938.68	612.63	326.05	1.53	1.28	285	307	258 (16)	7949 (69)
36	1525.99	995.25	530.74	1.53	1.28	410	382	348 (17)	6408 (62)
37	2464.13	1611.45	852.68	1.53	1.28	950	456	414 (18)	10592 (75)
38	3927.64	2577.39	1350.25	1.52	1.28	1223	546	504 (18)	34431 (74)
39	6030.77	4031.98	1998.79	1.50	1.26	674	633	608 (18)	29996 (72)

Table 7.1: Detailed experimental results.

Solver	**Solved (Competition)**	**Solved (Benchmarks)**	**Unique**
Simple BDD Solver (2)	195	189	10 *(6)*
AbsSynthe (seq2)	187	139	2
Simple BDD Solver (1)	185	175	
AbsSynthe (seq3)	179	134	
Realizer (sequential)	179		
AbsSynthe (seq1)	173	139	1
Demiurge (D1real)	139	136	5 *(2)*
Aisy	98		
Unbounded Synthesis		*103*	*12*

Table 7.2: Synthesis Competition 2015 Results

1. `6s216rb0_c0to31`
2. `cnt30y`
3. `driver_a10n`
4. `driver_a8n`
5. `driver_b10y`
6. `driver_b8y`
7. `driver_c10n`
8. `stay18y`
9. `stay20n`
10. `stay20y`
11. `stay22n`
12. `stay22y`

Table 7.3: Instances uniquely solved by our approach

8 Conclusion

List of Figures

Bibliography

Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. *Symbolic Reachability Analysis Based on SAT-Solvers*, pages 411–425. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

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.

Abdelwaheb Ayari and David A. Basin. QUBOS: deciding quantified boolean logic using propositional satisfiability solvers. In *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*, pages 187–201, 2002.

Armin Biere. *Resolve and Expand*, pages 59–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

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.

Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Könighofer, and Florian Lonsing. Sat-based methods for circuit synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 31–34, 2014.

Bart Bogaerts, Tomi Janhunen, and Shahab Tasharrofi. Solving QBF instances with nested SAT solvers. In *Beyond NP Workshop 2016 at AAAI'16*, 2016.

Aaron R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract*

Interpretation, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin, and Ocan Sankur. AbsSynthe: abstract synthesis from succinct safety specifications. In *Workshop on Synthesis, SYNT*, pages 100–116, 2014.

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.

H.K. Büning, M. Karpinski, and A. Flögel. Resolution for quantified boolean formulas. *Inf. Comput.*, 117(1):12–18, February 1995.

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.

Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 262–267, 1998.

CBS News, 2010. Toyota 'Unintended Acceleration' Has Killed 89. *CBS News*, 2010.

Ting-Wei Chiang and Jie-Hong R. Jiang. Property-directed synthesis of reactive systems from safety specifications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 794–801. IEEE Press, 2015.

Alonozo Church. Logic, arithmetic and automata. In *International Congress of Mathematicians*, pages 23–35, 1962.

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.

Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22 (2), 1997.

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

Niklas Eén, Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. SAT-based strategy extraction in reachability games. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3738–3745, 2015.

Uwe Egly, Florian Lonsing, and Magdalena Widl. Long-distance resolution: Proof generation and strategy extraction in search-based QBF solving. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR*, volume 8312 of *Lecture Notes in Computer Science*, pages 291–308. Springer, 2013.

Rüdiger Ehlers. Symbolic bounded synthesis. *Formal Methods in System Design*, 40(2):232–262, 2012.

Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011.

Bernd Finkbeiner and Swen Jacobs. Lazy synthesis. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 219–234, 2012.

Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.

Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Learning for quantified boolean logic satisfiability. In *Proceedings of the Eighteenth AAAI Conference on Artificial Intelligence, July 28 – August 1, 2002, Edmonton, Alberta, Canada.*, pages 649–654, 2002.

Alexandra Goultiaeva, Vicki Iverson, and Fahiem Bacchus. Beyond cnf: A circuit-based qbf solver. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, pages 412–426, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

Alexandra Goultiaeva, Allen Van Gelder, and Fahiem Bacchus. A uniform approach for generating proofs and strategies for both True and False QBF formulas. In *IJCAI*, pages 546–553, Barcelona, Spain, July 2011.

Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. *SAT-Based Image Computation with Application in Reachability Analysis*, pages 391–408. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

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.

Mikoláš Janota and Joao Marques-Silva. Solving qbf by clause selection. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 325–331, 2015.

MikolÃąs Janota and Joao Marques-Silva. On propositional QBF expansions and q-resolution. In *SAT*, pages 67–82, Helsinki, Finland, July 2013.

William Klieber, Samir Sapra, Sicun Gao, and Edmund Clarke. A non-prenex, non-clausal qbf solver with game-state learning. In *International Conference*

on Theory and Applications of Satisfiability Testing, pages 128–142. Springer, 2010.

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.

Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. A SAT-based counterexample guided method for unbounded synthesis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 364–382, 2016.

Florian Lonsing and Armin Biere. *Integrating Dependency Schemes in Search-Based QBF Solvers*, pages 158–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

K. L. McMillan. *Interpolation and SAT-Based Model Checking*, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

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.

Ken L. McMillan. *Applying SAT Methods in Unbounded Symbolic Model Checking*, pages 250–264. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

Andreas Morgenstern, Manuel Gesell, and Klaus Schneider. *Solving Games Using Incremental Induction*, pages 177–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

Nina Narodytska, Alexander Legg, Fahiem Bacchus, Leonid Ryzhyk, and Adam Walker. Solving games without controllable predecessor. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 533–540, 2014.

Patricia Parrish. Bookout v. Toyota Motor Corporation. District Court, Oklahoma County, OK, USA, 2013.

Florian Pigorsch and Christoph Scholl. Exploiting structure in an aig based qbf solver. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1596–1601, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

Florian Pigorsch and Christoph Scholl. An aig-based qbf-solver using sat for preprocessing. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 170–175, New York, NY, USA, 2010. 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.

Markus N. Rabe and Leander Tentrup. Caqe: A certifying qbf solver. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 136–143, 2015.

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.

Adam Walker and Leonid Ryzhyk. Predicate abstraction for reactive synthesis. In *Formal Methods in Computer-Aided Design FMCAD*, pages 219–226, 2014.

Poul F. Williams, Armin Biere, Edmund M. Clarke, and Anubhav Gupta. *Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking*, pages 124–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

Lintao Zhang and Sharad Malik. *Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation*, pages 200–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.