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

September 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 Swarat Chaudhuri and Azadeh Farzan, editors, *Proceedings of the 28th International Conference on Computer Aided Verification*, volume 9780 of *Lecture Notes in Computer Science*, pages 364–382, Toronto, ON, Canada, 2016. Springer

- Niklas Eén, Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. SAT-based strategy extraction in reachability games. In Blai Bonet and Sven Koenig, editors, *Proceedings of the 29th Conference on Artificial Intelligence*, pages 3738–3745, Austin, TX, USA, 2015. AAAI Press

- Nina Narodytska, Alexander Legg, Fahiem Bacchus, Leonid Ryzhyk, and Adam Walker. Solving games without controllable predecessor. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 533–540, Vienna, Austria, 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. An abstraction restricts the opponent to a set of moves and the algorithm existentially checks whether a strategy can be defeated by any combination of that restricted subset. Thus the universal quantification of opponent actions is handled by carefully expanding an existential formula only when required.

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 win-

ning 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,
- an initial state $s_0 \in S$,
- 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**ϕ - ϕ is true in the next state of the execution,
- **F**ϕ - Eventually (finally) ϕ will be true, and
- **G**ϕ - 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**ϕ iff for some state s', $R(s, s') \wedge M, s' \models \phi$.
- $M, s_0 \models$ **F**ϕ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi))$.
- $M, s_0 \models$ **G**ϕ 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**ϕ - ϕ is true on all paths
- **E**ϕ - 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', $R(s, s') \rightarrow M, s' \models \phi$.
- $M, s_0 \models \mathbf{A}[\phi_1 U \phi_2]$ iff for all paths $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.
- $M, s_0 \models \mathbf{E}[\phi_1 U \phi_2]$ iff for some path $(s_0, s_1, ...)$, $\exists i(i \geq 0 \wedge (M, s_i \models \phi_2) \wedge \forall j(j \geq 0 \wedge g < i \rightarrow (M, s_j \models \phi_1)))$.
- $(M, s \models AF\phi) \Leftrightarrow (M, s \models \mathbf{A}[\top U \phi])$
- $(M, s \models EF\phi) \Leftrightarrow (M, s \models \mathbf{E}[\top U \phi])$
- $(M, s \models AG\phi) \Leftrightarrow (M, s \models \neg EF(\neg\phi))$
- $(M, s \models EG\phi) \Leftrightarrow (M, s \models \neg AF(\neg\phi))$

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 $\mathbf{E}(GF\phi)$, 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.

Bounded model checking is another approach that searches for counterexample traces of a certain length to safety properties using a SAT solver. This form of model checking is the inspiration for the bounded approach to synthesis presented in this thesis and is covered in greater detail in Section 2.4.1.

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.

2.2.2 Symbolic Model Checking

The model checking approaches described above involve a computationally expensive explicit exploration of the state space of the system. 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 Kripke structure.

Consider the Kripke structures used in the previous sections. Without loss of generality, we may replace the set of states with a set of boolean variables. A single state is now a valuation to those variables and a set of states may

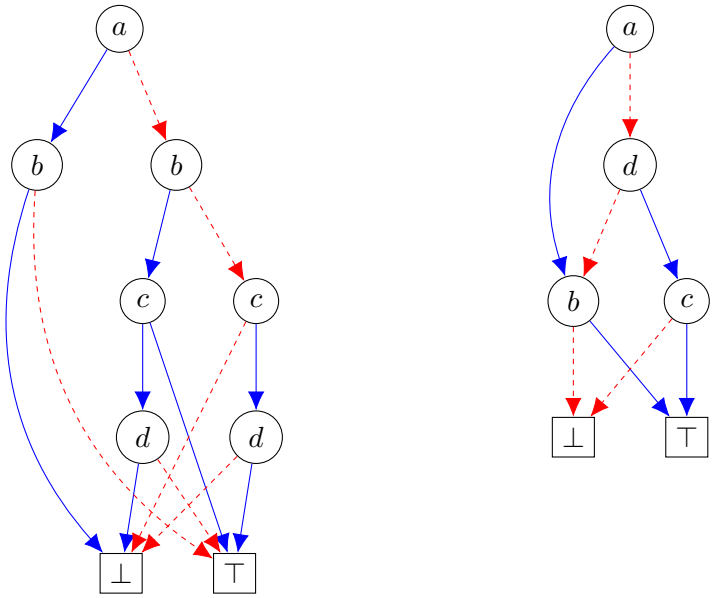


Figure 2.1: BDDs for $(\neg a \vee b) \wedge (a \vee c \vee \neg d) \wedge (b \vee d)$.
Solid blue transitions are 1, dashed red transitions are 0.

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 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. An efficient use of BDDs for model checking uses these operations for a fixed point computation that computes sets of states that satisfy a CTL property.

2.2.3 Fixed point calculations

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.

We can use μ-calculus to characterise CTL formulas as fixed point computations. For example, the formula $\mathbf{EG}\psi$ can be characterised with a greatest fixed point $\nu Z.\psi \wedge \mathbf{EX}Z$. With this characterisation the formula can be checked by iterative executions of $\mathbf{EX}Z$, which we call the *image* of Z. The image of a set of states is the set of all possible successor states. Likewise we say that the *pre-image* of a set of states is the set of all ancestor states. Both operations can be efficiently implemented on symbolic BDD representations of sets of states to check CTL formulas by computing fixed points. The fixed point can be computed forwards from the initial state using image operations or backwards from a target set using pre-image operations.

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 $G = (S, U, C, \delta, s_0)$. 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,
- U is a finite alphabet of *uncontrollable* actions,
- C is a finite alphabet of *controllable* actions,
- $\delta : S \times U \times C \to S$ 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 U and the system chooses a controllable action from C. We consider only deterministic games where $\delta(s, u, c, s'_1) \wedge \delta(s, u, c, s'_2) \rightarrow (s'_1 = s'_2)$. We modify the notion of a run to suit games: $(s_0, u_0, c_0), (s_1, u_1, c_1), ...(s_n, u_n, c_n)$ where $\forall i[i \geq 0 \rightarrow \delta(s_i, u_i, c_i, s_{i+1})]$.

In addition to the game structure itself we define a game *objective* ψ given by an LTL formula. We say that a run is *winning* for the controller iff the run satisfies the objective. The game is zero-sum, therefore a run is winning for the environment in the dual case where the objective ψ is not true. 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* $\pi^c : S \times U \rightarrow C$ 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* $\pi^e : S \rightarrow U$ 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 $\mathbf{G}\phi$ 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 using a fixed point computation similar to symbolic model checking of CTL properties as described above [Asarin et al., 1995].

Whilst the LTL fragment that is solvable via a safety game is simple, fixed point computations can also be used to solve the much more expressive generalised reactivity fragment of LTL [Piterman et al., 2006]. Safety synthesis can be seen as the first step on the path to more useful yet practical synthesis techniques.

2.3.2 Symbolic Game Solving

Solving a game symbolically is similar to symbolic model checking. Without loss of generality we may replace the sets of explicit states and actions that define the game with sets of boolean variables. We have $G = (\mathcal{S}, \mathcal{U}, \mathcal{C}, \delta, s_0)$ where $\mathcal{S}$, $\mathcal{U}$, and $\mathcal{C}$ are sets of boolean state and action variables. Then $\delta : 2^{\mathcal{S}} \times 2^{\mathcal{U}} \times 2^{\mathcal{C}} \rightarrow 2^{\mathcal{S}}$ is a boolean function that defines the transition relation of the game. $s_0 \in 2^{\mathcal{S}}$ is the initial state of the game.

Symbolic algorithms for solving safety games focus of determining sets of states from which a player 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 as

$$UPre(X) = \{s \mid \exists u \forall c \forall s' (\delta(s, u, c, s') \implies s' \in X)\}$$

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. In terms of μ-calculus, the environment's winning region with respect to an error set E is given by the least fixed point

of the uncontrollable predecessor of E. The complement of the environment's winning region gives the safe region for the controller:

$$SAFE(E) = \neg(\mu Y.UPre(Y \vee E))$$

As in model checking, fixed point calculations on sets of states may be efficiently implemented using binary decision diagrams to represent sets of game states. Synthesis algorithms implemented in this way are able to scale to games with large state spaces.

2.3.3 Abstraction

When the state space is very large, as it can be in many real world systems, symbolic representation is an insufficient optimisation. 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. This is known as predicate abstraction [Graf and Saidi, 1997] and is just one of several possible ways to abstract a specification.

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.

The counterexample guided abstraction methodology I describe in this thesis does not reduce the state space of the game in the same way that a technique like predicate abstraction does. In my approach a CEGAR loop is

used in a different way that is complementary to existing abstraction methods. For instance, predicate abstraction may be applied to a specification before it is solved via the algorithms presented in this thesis.

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*.The algorithm is described in further detail in the next chapter.

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 been 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 set of safe states 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 that violates the model's specification.

BMC considers the validity of CTL* formulas in Kripke structures in which the length of a run is bounded to k. The authors of the procedure provide a semantics for the translation of CTL* formulas on bounded models, via LTL, to satisfiability constraints. For example, consider a safety property $AG\phi$ where ϕ is a propositional formula. 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 from the initial state s_0 like so: $s_0 \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$. The transition relation describes all valid successor states of the current state, and k repeated applications of R describes all valid runs of length k. The LTL formula is similarly translated into a formula $\bigvee_{i=0}^{k} \neg\phi \in L(s_i)$. The SAT query is satisfiable when there is a path of length k in the Kripke structure $(s_0, s_1, ..., s_k)$ such that $\neg\phi$ holds in some state s_i.

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 of traces SAT can instead be used to symbolically compute reachable states of the system. The drawback is that although BMC is sound, i.e. any counterexample is a true counterexample, it is not complete with respect to the unbounded system unless a sufficient bound is used. For safety properties the diameter of the system gives a tight sufficient upper bound for BMC although it is difficult to compute.

The approach taken to realisability described in Chapter 4 is inspired by BMC and uses a similar approach to replace BDDs with SAT queries. In my approach a bound is placed on the number of rounds in a safety game. The method of unrolling the transition relation into a SAT query is adapted to games by the addition of branching to encode a partial strategy for one player. Similar to the use of SAT to search for counterexample traces in BMC, a SAT solver is used to check for an opponent spoiling strategy.

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 image operators computed during the fixed point calculation require quantifier eliminations 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 this 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. As introduced in Chapter 2, a fixed point algorithm can be used to compute the entire set of reachable states by repeated application of the image operator. When the set of states returned by the image operation is equivalent to the input states, i.e. the procedures encounters a fixed point, the entire set of reachable states is known.

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 transition and $k-1$ remaining transitions 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 $k-1$ steps.

The algorithm (given in Algorithm 1) takes a set of initial states $I \in 2^S$ where S is a set of boolean state variables, a transition relation $\delta : 2^S \times 2^S$, a set of final states $E \in 2^S$, and a bound k. The algorithm maintains a set of states R that overapproximates the reachable states from I. The approximation is initialised with I itself and each iteration updates the approximation with the

Algorithm 1 FiniteRun: Determines the existence of a finite run from I to E

function FINITERUN(I, δ, E, k)
  if $I \wedge E$ **then return** `true`
  $R \leftarrow I$
  loop
    $A \leftarrow R(s_0) \wedge \delta(s_0, s_1)$
    $B \leftarrow \left(\bigwedge_{1 \leq i \leq k} \delta(s_i, s_{i+1})\right) \wedge \left(\bigvee_{1 \leq i \leq k} E(s_i)\right)$
    if SAT($A \wedge B$) **then**
      if $R = I$ **then return** `true` **else** FINITERUN($I, \delta, E, k+1$)
    else
      $\mathcal{I} \leftarrow$ INTERPOLATE(A, B)
      $R' \leftarrow \mathcal{I}(s_1)$
      if $R' \implies R$ **then return** `false`
      $R \leftarrow R \vee R'$
    end if
  end loop
end function

addition of an interpolant constructed in the manner described above. If it is possible to reach a final state from I in k steps it will be detected by the SAT query on line 7 and the algorithm returns that a finite run exists. If it is not possible to reach a final state from I but it is possible from an approximation of the reachable set then a larger bound is required to determine if a run is actually possible. Otherwise, if the approximation reaches a fixed point then it forms an inductive invariant of the system and the algorithm is able to return that no run is possible.

3.2.5 Properly Directed Reachability (PDR)

More recently an approach was proposed that checks safety properties without unrolling the transition relation of the system [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 $0, 1, ..., k$ steps. The sequence is extended when $F_k(s) \wedge \delta(s, s') \rightarrow P'(s')$ 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 this thesis 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 this thesis. 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, 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 an error state in some number of game rounds. A state is added to the overappoximation of states that are envi-

ronment 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], 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 takes 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 (see Section 2.4.3). In accordance with the additional complexity of quantification (see Chapter 2) 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 this section we survey existing techniques for QBF solving, including the algorithm of [Janota et al., 2012] that inspired the bounded realisability algorithm presented in the next chapter.

Approaches to solving QBFs are split into two main categories: solvers that *search* for solutions with a modified DPLL algorithm [Cadoli et al., 1998] or solvers that *expand* the formula into an existential-only SAT problem [Ayari and Basin, 2002]. Research in the former category is focused mostly on computational learning to prune the search tree. Early work in this area identified that both conflict clauses and satisfying assignments (or *cubes*) can be learned in a QBF search [Giunchiglia et al., 2002; Zhang and Malik, 2002]. In the latter category research is focused on ensuring that expansion does not explode the size of the formula. In the following sections we review the state of the art in both of these categories of QBF solvers.

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 their 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], 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 Chapter 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 constructing 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. In a partial expansion only a subset of assignments to quantified vari-

ables are used to expand the formula. In other words, the player corresponding to those variables is restricted to a subset of actions. If the current player cannot win an abstract game in against a restricted opponent then it cannot win in the concrete game. So the abstraction can be used to find a satisfying assignment to its own variables with an efficient SAT query using the partially expanded formula. From that assignment, a candidate strategy is formed by taking the assignment to the first block in the quantifier prefix. The candidate is then checked by recursively calling the solver on the suffix of the formula. This effectively replaces the player's current action with its candidate 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 2.

Algorithm 2 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
```

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 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

In this chapter I reviewed research from the fields of model checking, synthesis, and QBF that are related to the work I present in this thesis. Below I summarise the most important of these.

- Bounded model checking is a technique used to check the correctness of a system by searching for execution traces allowed by the model that serve as counterexamples to its specification. It is able to do so efficiently by constructing a SAT query that is satisfiable when a counterexample trace of a certain length exists.

- Unbounded model checking optimises bounded model checking by proving the nonexistence of a trace of any length by constructing a set of reachable states. This can be achieved by either modifying the SAT decision procedure to carefully enumerate states or by using Craig interpolation to approximate the set.

- Properly directed reachability also uses interpolation to approximate reachable states but does so by incrementally constructing a series of inductive lemmas for the system.

- There are several existing approaches to synthesis with SAT. Bounded synthesis places an upper limit on the size of implementation and lazy synthesis extends this by guiding the search for an implementation with counterexamples. There are also approaches that generalise clause learning for unbounded model checking and properly directed reachability into synthesis.

- The natural extension of bounded model checking to synthesis replaces SAT queries with QBF. Recent research has suggested that efficient QBF solvers should focus on reconstructing and exploiting information about the problem. Additionally, approaches to QBF that use dueling SAT solvers have seen success.

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 the case of realisability, the existence of a trace that reaches an error state does not guarantee that the specification is unrealisable. Instead we are interested in player *strategies*. A controller strategy must avoid the error states for all possible environment actions. Likewise, an environment 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 does not guarantee unbounded realisability. 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 $G = (\mathcal{S}, \mathcal{U}, \mathcal{C}, \delta, s_0, E)$. $S, \mathcal{U}$, 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 $\delta : 2^S \times 2^{\mathcal{U}} \times 2^{\mathcal{C}} \rightarrow 2^{\mathcal{S}}$ 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. A controller strategy is a function $\pi^c : 2^{\mathcal{S}} \times 2^{\mathcal{U}} \rightarrow 2^{\mathcal{C}}$, i.e. a mapping from states and environment actions to controller actions. We say that a strategy is winning if all runs in which the controller chooses actions according to π^c avoid error states. If π^c is a partial function that defines a mapping for a subset of $2^{\mathcal{S}} \times 2^{\mathcal{U}}$ to $2^{\mathcal{C}}$ then we call π^c a partial strategy.

Realisability is the problem to determining the existence of a winning controller strategy. The following quantified formula may be used to solve realisability of a safety game bounded to k game rounds:

$$\forall \mathcal{U}_k \exists \mathcal{C}_k \forall \mathcal{U}_{k-1} \exists \mathcal{C}_{k-1} \ldots \forall \mathcal{U}_0 \exists \mathcal{C}_0$$
$$(\neg E(\mathcal{S}_k) \wedge \delta(\mathcal{S}_k, \mathcal{U}_k, \mathcal{C}_k, \mathcal{S}_{k-1}) \wedge \ldots \wedge \neg E(\mathcal{S}_1) \wedge \delta(\mathcal{S}_1, \mathcal{U}_1, \mathcal{C}_1, \mathcal{S}_0) \wedge \neg E(\mathcal{S}_0)).$$

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. Hence it is 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 the remainder of this chapter I describe an adaptation of the counterexample guided QBF algorithm that takes advantage of this information.

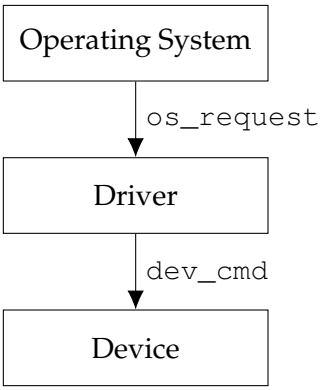


Figure 4.1: Structure of device driver example

4.1.1 Example

We introduce an example to assist an intuitive explanation of the algorithm. Consider a model of a simple storage device driver. The operating system makes requests of the driver to write or read data to or from the device. The structure of the model is shown in Figure 4.1. 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 $G = (\mathcal{S}, \mathcal{U}, \mathcal{C}, \delta, s_0)$. The structure for our example is:

- $\mathcal{S} = \{\texttt{request}, \texttt{error}\}$. The game consists of two boolean variables to denote the current request from the OS to the driver, and whether an error has occurred. We use $\texttt{request} = 0$ to represent a `read` and $\texttt{request} = 1$ for `write`.
- $\mathcal{U} = \{\texttt{os\_request}\}$. The uncontrollable actions consist 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` ($\texttt{dev\_cmd} = 0$) or a `write` ($\texttt{dev\_cmd} = 1$).
- The transition relation δ is defined by the following formulas:

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

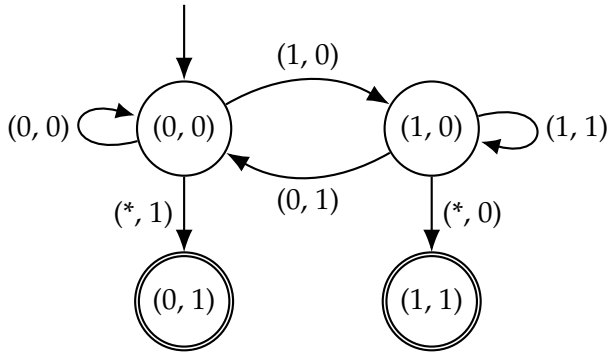


Figure 4.2: State automata representation of δ in Example 1. Nodes are labelled by the tuple (`request`, `error`). Edges are labelled with uncontrollable and controllable actions: (`os_request`, `dev_cmd`). A star indicates that the transition occurs on both a 0 and a 1. Transitions from error states are elided for simplicity.

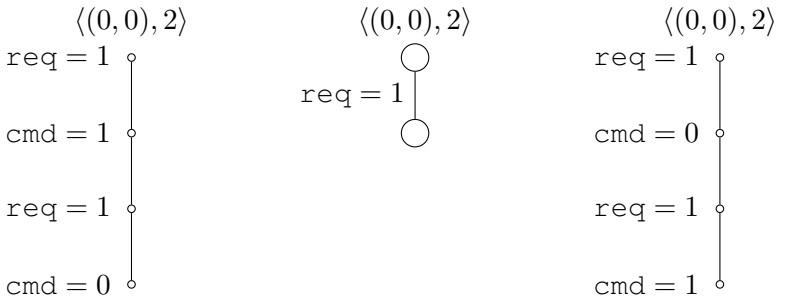
$$\texttt{err'} \leftarrow \texttt{request} \neq \texttt{dev\_cmd}$$

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

- $s_0 = (\texttt{request} = 0 \wedge \texttt{err} = 0)$. 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 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 4.3b shows an example abstract game tree restricting the environment (abstract game trees restricting the controller are similar). The root of the tree is labelled with an initial state and a game length. In the abstract game, the



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

Figure 4.3: Execution of bounded realisability on the example. Here we use abbreviations `req` for `os_request` and `cmd` for `dev_cmd`. Root nodes are labelled with initial state (`os_request`, `dev_cmd`) and a game length.

controller can freely choose actions whilst the environment is required to pick actions from the tree. After reaching a leaf, the environment continues playing unrestricted. The tree in Figure 4.3b 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. Effectively, FINDCANDIDATE employs cooperation as a heuristic for optimistically discovering candidate strategies or alternatively quickly discovering useful counterexamples that speed up the refinement loop. The heuristic is based on the observation that most real world systems to which synthesis may be applied are designed to allow the implementation of an efficient controller. Figure 4.3a shows a trace through the example game that reaches an error state.

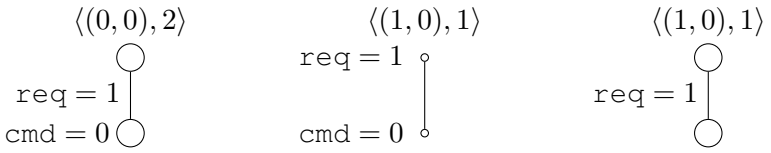
The trace informs us that by playing the actions contained in the trace it is possible for the environment to reach an error state. From this we conjecture that the first action in the trace is a reasonable choice for the first move in

the environment's winning strategy. So we construct a candidate strategy in which the environment plays $\texttt{os\_request} = 1$ 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 4.3b). 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 4.3c shows a trace in which the controller counters the environment by playing $\texttt{dev\_cmd} = 0$ in the first round to match the initial state of $\texttt{request} = 0$.

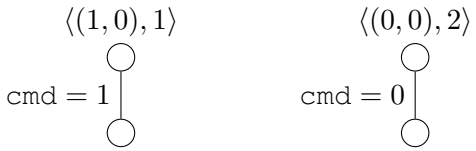
We can define a partial strategy for an abstract game by labelling the nodes in the tree to define the actions a player should choose against the opponent's actions in the edges of the tree. The trace contains controller actions that can be used to form a counterexample partial strategy (Figure 4.4a). 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 there is a unique state $(\texttt{request} = 1 \wedge \texttt{err} = 0)$ reachable by playing the actions in the combination of AGT and counterexample strategy. We now solve an abstract game with a bound of 1 from this state in order to determine which action the environment should select for the second game round.

This new game is solved via a recursive call to the algorithm. First a trace to an error state is found (Figure 4.4b) and an environment candidate strategy is constructed (Figure 4.4c) from the first environment action in the trace. Then a counterexample trace to the environment's strategy is found in which the controller chooses correctly to play $\texttt{dev\_cmd} = 1$. 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 4.4d).

The environment is now playing against a restricted opponent. In this case there is no possible action that the environment can take to reach an error state. This can be seen in the state machine for the game in Figure 4.2. None



(a) Partial strategy (b) Trace to error (c) AGT with controller actions



(d) Refined AGT (e) Refined AGT

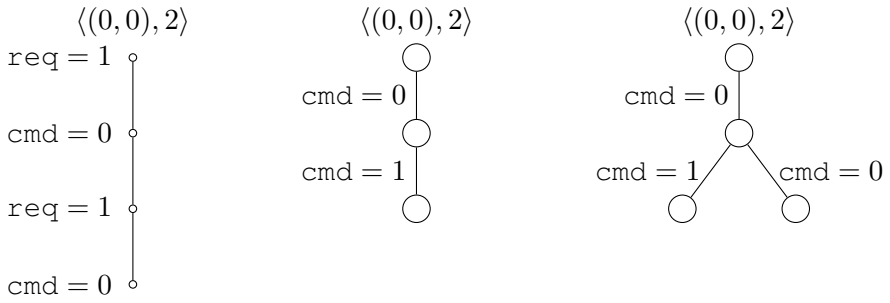
Figure 4.4: Continued example algorithm execution

of the outgoing transitions from $(1, 0)$ with the controller choosing to play $\texttt{dev\_cmd} = 1$ lead to an error state in one step. 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 4.3b can be defeated by the partial controller strategy in Figure 4.4a. 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 4.4e. Note that a trace reaching an error in which the environment plays $\texttt{os\_request} = 1$ is still possible in this abstract game (Figure 4.5a). The algorithm without optimisation may 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 4.5b). On this game abstraction the candidate is blocked because the trace must include $\texttt{dev\_cmd} = 1$ in the second round.

As a result the environment must choose a different action for the first



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

Figure 4.5: Continued example algorithm execution (2)

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 4.5c) the environment has no winning trace and therefore the algorithm terminates and returns realisable for the controller. The final game tree serves as a *certificate tree* that proves the nonexistence of an environment strategy. In the next chapter I will show how a controller strategy can be extracted from a certificate tree.

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 to one of the players. For example, in Figure 4.5c the controller is restricted to playing `dev_cmd` $= 0$ in the first round and either `dev_cmd` $= 0$ or `dev_cmd` $= 1$ in the second. 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. Figure 4.6 shows how HEIGHT is calculated, n_1 has a height of two despite being one edge away from a leaf because the tree does not reach the end of a three round game. 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

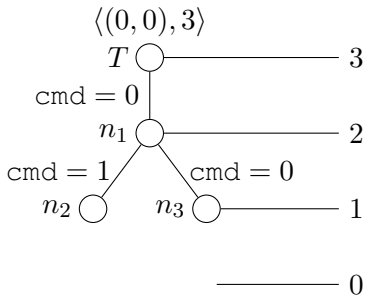


Figure 4.6: Height of an AGT node

and n'.

Given an environment (controller) abstract game tree T a *partial strategy* $Strat : \text{NODES}(T) \to 2^{\mathcal{C}}$ ($Strat : \text{NODES}(T) \to 2^{\mathcal{U}}$) labels each node n of the tree with the controller's (environment's) action to be played in the game round corresponding to $\text{HEIGHT}(k, n)$. Figure 4.4a shows a partial strategy constructed during the example execution. In the example an action for the controller ($\texttt{dev\_cmd} = 0$) is defined as the action to play in the first round of the game against the environment's first action ($\texttt{os\_request} = 1$).

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 if 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 we achieve the goal of directing the search towards strong strategies and counterexamples.

Algorithm 3 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
```

The full procedure is illustrated in Algorithms 3 to 5. 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 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 by solving new games beginning at the outcome in each leaf of the AGT after applying the candidate strategy. The new games are solved by a recursive call to SOLVEABSTRACT, which now plays on behalf of the opponent. The dual

solver is searching for a continuation of the current game that ensures that the opponent always wins.

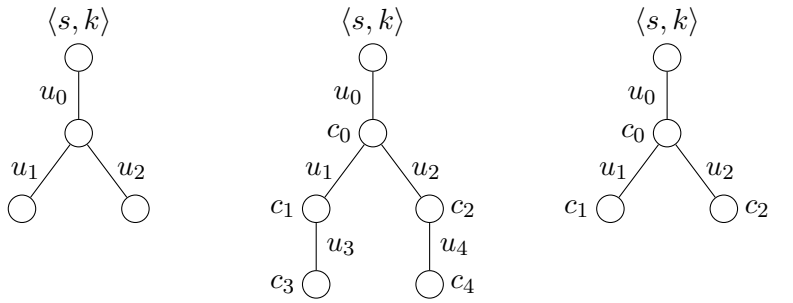
Algorithm 4 Find a candidate strategy

17: **function** FINDCANDIDATE(p, s, k, T)
18: $\hat{T} \leftarrow$ EXTEND(T) ▷ Extend the tree with arbitrary actions
19: **if** $p = \texttt{cont}$ **then**
20: $f \leftarrow$ TREEFORMULA($k, \hat{T}$)
21: **else**
22: $f \leftarrow \overline{\text{TREEFORMULA}}(k, \hat{T})$
23: **end if**
24: $sol \leftarrow \text{SAT}(s(\mathcal{S}_{\hat{T}}) \wedge f)$
25: **if** $sol = \texttt{unsat}$ **then**
26: LEARN(p, s, k, T, f) ▷ This line is enabled in an optimisation
27: **return** NULL ▷ No candidate exists
28: **else**
29: ▷ Return partial strategy for T
30: **return** $\{\langle n, c\rangle \mid n \in \text{NODES}(T), c = sol(n)\}$
31: **end if**
32: **end function**

Algorithm 5 Verify a candidate strategy

33: **function** VERIFY($p, s, k, T, cand$)
34: **for** $l \in$ LEAVES(T) **do**
35: $\langle k', s'\rangle \leftarrow$ OUTCOME($s, k, cand, l$) ▷ Get bound and state at leaf
36: **if** $k' = 0$ **then continue**
37: **if** $p =$ CONT **then**
38: $T' \leftarrow \emptyset$
39: **else**
40: $T' \leftarrow \{cand(l)\}$
41: **end if**
42: ▷ Solve for the opponent
43: $a \leftarrow$ SOLVEABSTRACT(OPPONENT(p), s', k', T')
44: **if** $a \neq \texttt{NULL}$ **then return** $\langle \texttt{true}, l, a\rangle$ ▷ Return counterexample
45: **end for**
46: **return** $\langle \texttt{false}, \emptyset, \emptyset\rangle$ ▷ There was no counterexample
47: **end function**

If the dual solver can find no spoiling strategy at any of the leaves then the candidate contains the prefix of a winning strategy for the abstract game. Otherwise, VERIFY returns the action used by the opponent after the leaf of



(a) Original AGT (b) Refined AGT with candidate strategy (c) Projected strategy

Figure 4.7: Projection of a candidate strategy

the AGT in a spoiling strategy. The abstract game is refined by appending this action to the corresponding leaf in T in line /refline:callVerify.

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 (see Figure 4.7). 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 formulate this heuristic as a SAT query such that any satisfying assignment encodes such a strategy. The query is constructed recursively by TREEFORMULA (for the controller) or $\overline{\text{TREEFORMULA}}$ (for the environment) in Algorithm 6.

The tree is first extended to the maximum bound with edges that are labeled with arbitrary opponent actions (Algorithm 4, line 18). For each node in the tree, new SAT variables are introduced corresponding to the state ($\mathcal{S}_T$) and action ($\mathcal{U}_T$ or $\mathcal{C}_T$) 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

ACTION(e). The state and action variables of node n are connected to successor nodes SUCC(n) by an encoding of the transition relation and constrained to the winning condition of the player.

Algorithm 6 Tree formulas for Controller and Environment

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

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

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

$$\bigvee_{\langle e,n\rangle \in \mathrm{SUCC}(T)} (\delta(\mathcal{S}_T, \mathcal{U}_T, \mathcal{C}_e, \mathcal{S}_n) \wedge C_e = \mathrm{ACTION}(e) \wedge \overline{\text{TREEFORMULA}}(k, n))$$

  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.

Proposition 1. *Let T be an abstract game tree with edges labelled by opponent actions. If* FINDCANDIDATE$(p, s, k, T) =$ `NULL` *then there is no strategy for p to win a game bounded to k rounds from s.*

Proof. We assume that p is the controller, the proof for the environment is similar. The first line of FINDCANDIDATE constructs $\hat{T}$ by extending T to the full length of the game with arbitrary environment actions. The algorithm then proceeds to construct $f =$ TREEFORMULA$(k, \hat{T})$ and query a SAT solver for a satisfying assignment to $s(\mathcal{S}_k) \wedge f)$. The algorithm returns `NULL` when there is no satisfying assignment.

At every iteration of TREEFORMULA, a conjunction expands a universal quantifier into a subset of its possible values: the actions in the tree. Therefore, $s(\mathcal{S}_k) \wedge f$ is a partial expansion of the formula representing the existence of a controller strategy:

$$\forall \mathcal{U}_k \exists \mathcal{C}_k \forall \mathcal{U}_{k-1} \exists \mathcal{C}_{k-1} \ldots \forall \mathcal{U}_0 \exists \mathcal{C}_0$$
$$(\neg E(\mathcal{S}_k) \wedge \delta(\mathcal{S}_k, \mathcal{U}_k, \mathcal{C}_k, \mathcal{S}_{k-1}) \wedge \ldots \wedge \neg E(\mathcal{S}_1) \wedge \delta(\mathcal{S}_1, \mathcal{U}_1, \mathcal{C}_1, \mathcal{S}_0) \wedge \neg E(\mathcal{S}_0)).$$

The partial expansion is satisfiable when there is a corresponding assignment to $\mathcal{C}_k \ldots \mathcal{C}_1$ for every value in the subset of expanded environment variables. If there is not an assignment that satisfies the formula for *one* value to universal variables then clearly the formula is not satisfiable for *all* values. Hence if FINDCANDIDATE returns `NULL` there cannot exist a strategy for the bounded game. □

Theorem 1. *If there exists a strategy to a bounded game then* SOLVEABSTRACT *will return a certificate tree labelled with a partial strategy.*

Proof. Consider an abstract game tree T with all opponent actions enumerated, i.e T is a concrete game tree. The call to FINDCANDIDATE on line 2 produces a formula exactly equivalent to a full expansion of the quantified formula above. If there is a winning strategy then it will be found by the SAT query and returned as a labelling of the certificate tree T. The call to VERIFY amounts to a check that the outcome in every leaf of the tree is a winning state for the opponent. The computed strategy guarantees that no outcome is losing so VERIFY must return false and the algorithm terminates.

If T does not yet contain all opponent actions then VERIFY may return a spoiling strategy that is then appended to the tree. The call to VERIFY recursively calls SOLVEABSTRACT for the opponent, so if a spoiling strategy is found

it must be a valid opponent strategy against the candidate by Proposition 1. Any action appended to the tree must be one that is winning for the opponent against the candidate and so cannot already exist in the abstraction. Hence the refinement grows monotonically with each call to VERIFY. Thus, either VERIFY returns false and the algorithm terminates, or eventually abstraction is refined into a concrete game tree. A winning strategy for the concrete game must be winning in any abstraction so even if T is the empty abstraction VERIFY will return false in this case. □

4.2 Optimisations

The bounded realisability algorithm has an worst case running time that is exponential in the number of opponent actions 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 7). 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. Enabling the optimisation triggers a call to LEARN in line 26 of FINDCANDIDATE. 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 $>= k$. Instead of keeping an array we maintain a single set of losing states B^c and block the entire set in all rounds of the game. This means that

the algorithm is no longer complete: we can no longer find strategies that visit states in a round $< k$ after we discover that they are losing states for $>= k$. In practical use we are uninterested in these controller strategies since they are not winning for the unbounded game.

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. If the formula remains unsatisfiable for the generalised cube then the entire set of states must also be losing. 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]. The entire learning procedure is shown in Algorithm 8. LEARN takes a state s, and a tree formula f that was found to be losing for the player k at height k. First, literals are greedily removed from s and the resulting generalised cube is added to B^c or B^e.

Algorithm 7 Modified Tree Formulas with Bad State Avoidance

function TREEFORMULA(k, T)
  if HEIGHT$(k, T) = 0$ **then**
    return $\neg B^c(\mathcal{S}_T)$
  else
    return $\neg B^c(\mathcal{S}_T) \wedge$

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

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

$$\bigvee_{\langle e,n\rangle \in \text{SUCC}(T)} (\delta(\mathcal{S}_T, \mathcal{U}_T, \mathcal{C}_e, \mathcal{S}_n) \wedge C_e = \text{ACTION}(e) \wedge \overline{\text{TREEFORMULA}}(k, n))$$

  end if
end function

Algorithm 8 Learn an expanded cube of losing states

```
function LEARN(p, s, k, T, f)
    ŝ ← s
    for s ∈ S do
        sol ← SATWITHASSUMPTIONS(ŝ \ {s}, f)
        if sol = NULL then
            ŝ ← ŝ \ {s}
        end if
    end for
    if p = cont then
        B^c ← B^c ∨ ŝ
    else
        for i ∈ [0, ..., k] do
            B^e[i] ← B^e[i] ∨ ŝ
        end for
    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* candidate 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 in fewer game rounds. For the environment this is clearly analogous to reaching the error set sooner. For the controller it is less clear, we use states that have been learned to be losing for the environment for a particular game height. The intuition is that these states are more likely to be *safe*, i.e. belonging to the winning region of the controller.

Algorithm 9 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:      α ← ⊤
5:      cand ← SAT(s(S_T̂) ∧ f)
6:      for l ∈ LEAVES(T) do
7:          n ← ROOT(l)
8:          while HEIGHT(k, n) ≠ 0 do
9:              if p = cont then
10:                 α̂ ← B^e[HEIGHT(k, n)](S_n)
11:             else
12:                 α̂ ← B^e(S_n)
13:             end if
14:             sol ← SATWITHASSUMPTIONS(α ∧ α̂, s(S_T̂) ∧ f)
15:             if sol ≠ NULL then
16:                 α ← α ∧ α̂
17:                 cand ← sol
18:                 break
19:             end if
20:             n ← SUCC(n)
21:         end while
22:     end for
23:     return cand
24: end function
```

Algorithm 9 gives the pseudocode for the optimisation. First a formula f representing the abstract game is constructed in the same way as in FINDCANDIDATE. An additional constraint on the formula α is initialised to true. Then for a leaf l in the game tree the algorithm greedily attempts to construct a candidate strategy such that the highest possible predecessor of l is a known winning state. If a candidate is found then the constraint is added to α and the algorithm continues on to find candidates with constraints added to other branches of the tree. The solution to the SAT query with the least permissive constraint is returned to be used as a candidate strategy.

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 cooperated. 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 controllable predecessor used to construct a BDD explores all branches before progressing to the next game round, and bounded realisability explores traces through the full height of the bounded game before constructing branches in the abstraction. As with search, the best performing 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.

A QBF solver may use dependency analysis to determine a partial ordering on decision making in the search. The bounded realisability algorithm is based on partial expansions and not search but the way that partial expansions are made is effectively a decision. In the case of bounded realisability formulas analysis will most likely not reveal a dependency tree significantly different from the linear quantifier prefix itself. The formula is constructed by unrolling a transition relation that takes states and actions as input and outputs states for the next iteration. It may be possible that the relation contains independent subformulas and some actions and states can be decided in parallel but the formula also contains constraints on states variables in the form of error and learned states. Thus it is unlikely that any action from a subsequent game round may be decided before an action in the current game round. One aspect of the dependency tree is that state variables are decided entirely by a prefix of action variable assignments. This information is used in bounded realisability when the OUTCOME of a labelled game tree is used to create a subgame.

The algorithm takes advantage of a higher level of information about the original problem than an analysis of the circuit could provide. A generalised approach to learning in QBF detects cubes of satisfying assignments and conflict clauses so that they are not reconsidered in other branches of the search after backtracking. Every unsatisfiable formula produced by FINDCANDIDATE is effectively discovering either a satisfying assignment (when the environ-

ment loses) or a conflict clauses (when the controller loses). The specialised solver projects this information onto state variables and transmits the learned states between rounds of the game. This is only possible with the knowledge that the formula is constructed from an iterated unrolling of the transition relation of a game and it would be difficult to reproduce this level of learning 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 verifying programs 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 realisability problem is modelled as a game of opposing players. 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.

- $\mathcal{S} = \{\texttt{error}\}$. The only state is whether or not an error has occurred.
- $\mathcal{U} = \{\texttt{os\_request}\}$.
- $\mathcal{C} = \{\texttt{dev\_cmd}\}$.
- 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 domain of the action variables then it is easy to see the potential blow up. In Figure 4.8 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. In this particular example learning that the state $\texttt{err} = 0$ is losing for the environment a various game rounds will significantly reduce the size of the tree. This example can be trivially extended so that learning will no longer be helpful by introducing states that record the history of the game. This extension can be done in a way that ensures that even with cube generalisation a unique state is learned in every node of the tree and learning does not reduce the search space.

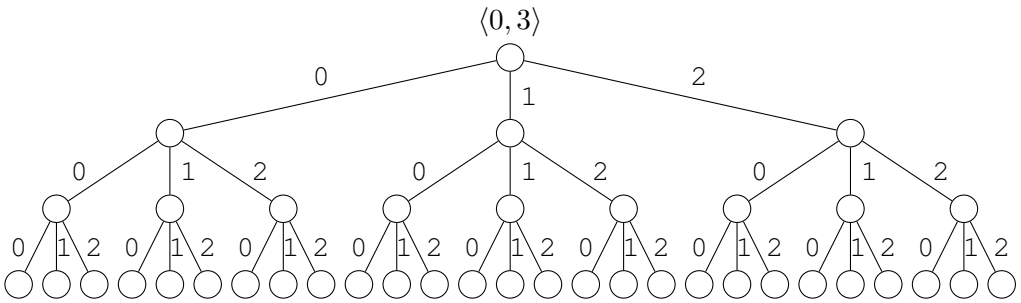


Figure 4.8: AGT with large branching factor

It should be noted that it is trivial to solve this example with a BDD solver, which can immediately prove that $\texttt{err} = 0$ 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 place 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.

- $\mathcal{S} = \{\texttt{error, bay0, bay1, bay2, bay3, timer}\}$.
- $\mathcal{U} = \{\texttt{load0, load1}\}$.
- $\mathcal{C} = \{\texttt{ship}\}$.

- The transition relation δ is now:

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

- $s_0 = (\texttt{error} = 0 \wedge \texttt{bay0} = 0 \wedge \texttt{bay1} = 0 \wedge \texttt{bay2} = 0$
  $\wedge\, \texttt{bay3} = 0 \wedge \texttt{timer} = 1)$

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 4.9, BDDs are not succinct when used to represent formulas that are in this enumeration of cubes style. If this problem was to scale the BDD would quickly consume a large amount of space.

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 presented here finds single paths through the tree that may be winning and checks them. Both approaches are efficient on different classes of specifications.

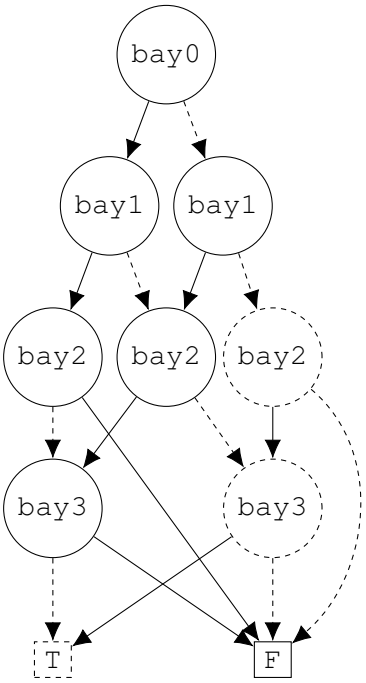


Figure 4.9: Environment winning region as a BDD.
Solid transitions are 1, dashed transitions are 0.

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 $\hat{y}_0, \hat{y}_1, \ldots, \hat{y}_i$ to existential variables $\hat{x}$ such that when substituting $\hat{x}$ for $f(\hat{y}_0, \hat{y}_1, \ldots, \hat{y}_i)$ 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. We simplify the problem by constructing a single function π that maps states and environment actions to controller actions. The game is deterministic, so an assignment to variables in the quantifier prefix corresponds to exactly one state and action pair. If we guarantee that all successors states reachable by playing according to the strategy in round k have a winning strategy defined by π for a game bounded to $k-1$ rounds then this function may be used as a Skolem function in every round of the game. Thus we are able to solve a simpler problem than Skolemisation of the entire QBF by taking advantage of the structure of bounded realisability.

5.1 Algorithm

Recall that a safety game is a tuple $(\mathcal{S}, \mathcal{U}, \mathcal{C}, \delta, s_0)$ where $\mathcal{S}$ is a set of boolean state variables, $\mathcal{U}$ a set of boolean environment action variables, $\mathcal{C}$ a set of boolean 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 states and environment actions to controller actions. For convenience we use $\mathcal{W} = \mathcal{S} \cup \mathcal{U}$ to denote the set of boolean variables that serve as input to the function defining a strategy.

In this chapter we will assume that the safety game is realisable and a winning strategy for the controller exists. However, the technique is also easily applied to unrealisable games to extract a spoiling strategy that is winning for the environment. 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.

For strategy extraction, we extend the notion of abstract game trees. A controller strategy defines a mapping from states and environment actions to controller actions. We label the root node of trees with a set $\sigma \subseteq 2^{\mathcal{W}}$ so that we may say that a tree is a certificate tree for a set of both states and environment actions. The tree computed by bounded realisability is labelled $s_0 \wedge \top$ and so is a certificate tree for all environment actions in the initial state.

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 $\mathcal{S} = \{\texttt{clk}, \texttt{curr}, \texttt{err}\}$

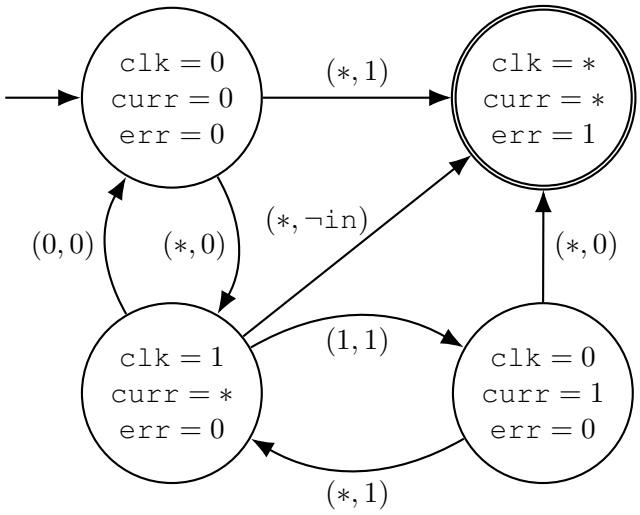


Figure 5.1: Transition relation of the running example

are the state variables of the game and contains the clock, the current value of the flip-flop, and an error bit respectively. The environment has a single data bit variable: $\mathcal{U} = \{\texttt{in}\}$ and the controller has the next value of the flip-flop: $\mathcal{C} = \{\texttt{next}\}$. The diagram shows δ as a deterministic finite state automaton with edges labelled by a tuple $(\texttt{in}, \texttt{next})$. We use $*$ as a wildcard value to simplify the presentation. The circuit described by the specification allows the environment to save a bit `in` into the flip-flop when the clock has a falling edge (`clk` transitions from 1 to 0). The controller must correctly set `next` so that `curr` always contains the correct data. The initial state, s_0, is $(\texttt{clk} = 0 \wedge \texttt{curr} = 0 \wedge \texttt{err} = 0)$ and the error set is given by $(\texttt{err} = 1)$.

Algorithm 10 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.

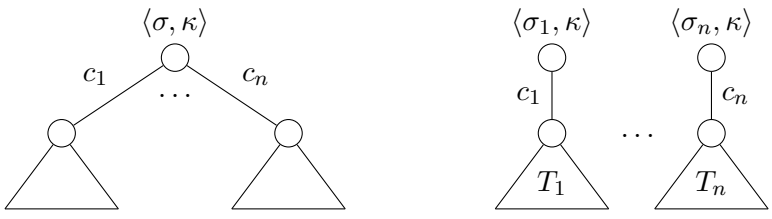
The GENLOCALSTRATS function recursively traverses the certificate tree T, starting from the root, computing local strategies in each node. The main operation of the algorithm, called PARTITION, splits (T, σ) into j tuples (T_i, σ_i), as shown in Figure 5.2. Each tree T_i is a copy of a single branch of T and the series $\sigma_1 \ldots \sigma_i$ forms a disjoint partitioning of σ, i.e. $\sigma_i \subseteq \sigma \subseteq 2^{\mathcal{W}}$. 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 for states and environment actions in σ_i.

Figure 5.3 illustrates how local strategies are generated from the winning abstract game tree returned by the game solver for our running example. Figure 5.3a shows T, the certificate tree of height 3 for the game. The algorithm starts at the root of the tree and the initial set of states and actions is given by $\sigma = s_0 \wedge \top = (\texttt{clk} = 0 \wedge \texttt{curr} = 0 \wedge \texttt{err} = 0)$. The game tree defines only one winning action in the root node, hence this action is winning in all states of σ and against all actions and no partitioning is required. We now compute the successor set reachable by playing action $\texttt{next} = 0$ against σ: i.e. we compute the set $\sigma' \subseteq 2^{\mathcal{W}'}$ such that $\exists\mathcal{S}\exists\mathcal{U}\exists\mathcal{C}(\delta(\mathcal{S},\mathcal{U},\mathcal{C},\mathcal{S}') \wedge \sigma \wedge (\texttt{next} = 0))$, which evaluates to $\sigma' = (\texttt{clk} = 1 \wedge \texttt{err} = 0)$.

Next, we descend down the tree and consider subtree T' and its initial set σ' (Figure 5.3b). We partition σ' into subsets $\sigma'_1 = (\texttt{clk} = 1 \wedge \texttt{err} = 0 \wedge \texttt{in} = 0)$ and $\sigma'_2 = (\texttt{clk} = 1 \wedge \texttt{err} = 0 \wedge \texttt{in} = 1)$ that are winning for the left and right subtrees of T' respectively, i.e., from $(\texttt{clk} = 1 \wedge \texttt{err} = 0)$ the controller must play action $\texttt{next} = 0$ when the environment plays $\texttt{in} = 0$, and $\texttt{next} = 1$ for $\texttt{in} = 1$. Consider the resulting subtrees T'_1 and T'_2 with initial sets σ'_1 and σ'_2 (Figure 5.3c). We compute successor states as before: $\sigma''_1 = (\texttt{clk} = 1 \wedge \texttt{curr} = 0 \wedge \texttt{err} = 0)$ and $\sigma''_2 = (\texttt{clk} = 1 \wedge \texttt{curr} = 1 \wedge \texttt{err} = 0)$, with corresponding subtrees T''_1 and T''_2 (Figure 5.3d). Both subtrees have one branch; hence the actions in those branches $\texttt{next} = 0$ and $\texttt{next} = 1$ are winning for σ''_1 and σ''_2 respectively.

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

Putting together fragments of the winning strategy computed above, we obtain the partial strategy shown below for this example. The compilation



(a) Before (b) After

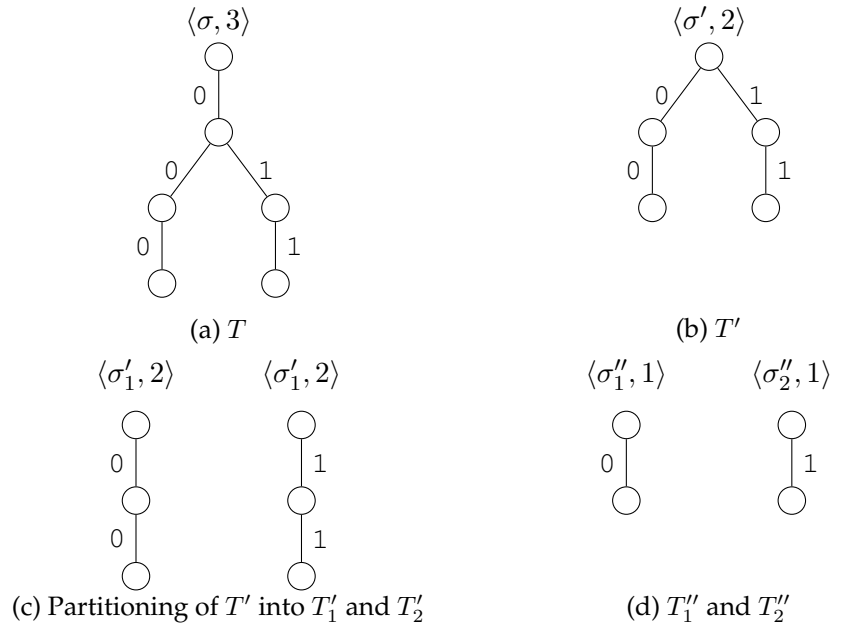
Figure 5.2: Partitioning

process must obey certain rules in order to correctly produce a winning partial strategy, the precise methodology is given below. A complete strategy can be constructed by asigning arbitrary actions to all other states as they are known to be unreachable by playing the partial strategy.

$$\pi(\texttt{clk} = 0 \wedge \texttt{curr} = 0 \wedge \texttt{err} = 0) = (\texttt{next} = 0)$$
$$\pi(\texttt{clk} = 1 \wedge \texttt{err} = 0 \wedge \texttt{in} = 0) = (\texttt{next} = 0)$$
$$\pi(\texttt{clk} = 1 \wedge \texttt{err} = 0 \wedge \texttt{in} = 1) = (\texttt{next} = 1)$$
$$\pi(\texttt{clk} = 0 \wedge \texttt{curr} = 1 \wedge \texttt{err} = 0) = (\texttt{next} = 1)$$

5.1.2 Partitioning game trees

The algorithm described so far involves two potentially costly operations: winning set partitioning and successor set computation. If implemented naïvely in a SAT based approach these operations can lead to unacceptable performance. Partitioning (σ, T) into a winning set for every $\langle e, n\rangle \in \text{SUCC}(T)$ requires computing all $\sigma_i \subseteq 2^{\mathcal{W}}$ that satisfies $(\sigma \wedge \mathcal{C}_e = c \wedge \neg\overline{\text{TREEFORMULA}}(k, n))$. Similarly, the c-successors of σ are computed by determining all $\sigma' \subseteq \mathcal{S}$ that satisfies $(\delta(\mathcal{S}, \mathcal{U}, \mathcal{C}, \mathcal{S}') \wedge \sigma \wedge \mathcal{C} = c)$. Both operations require elimination of



(a) T (b) T'

(c) Partitioning of T' into T'_1 and T'_2 (d) T''_1 and T''_2

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

Algorithm 10 Computing a winning strategy

```
1: function GenStrategy(T, k, σ)
2:     Strat ← GenLocalStrats(T, k, σ)
3:     return CompileStrat(Strat)
4: end function

5: function GenLocalStrats(T, k, σ)
6:     [(e_1, n_1), ..., (e_j, n_j)] ← succ(T)
7:     [(T_1, σ_1), ..., (T_j, σ_j)] ← partition(T, k, σ)
8:     Strat ← {(σ_i, action(e_i), height(k, T)) | i ∈ [1, ..., j]}
9:     for i = 1 to j do
10:        (T'_i, σ'_i) ← next(T_i, k, σ_i)
11:        Strat_i ← GenLocalStrats(T'_i, k − 1, σ'_i)
12:        Strat ← Strat ∪ Strat_i
13:    end for
14:    return Strat
15: end function
```

existential quantifiers from the formulas. To eliminate quantifiers with SAT requires an inefficient enumeration of satisfying assignments. The key insight behind our solution is that both operations can be efficiently approximated from the proof of unsatisfiability of the formula $\sigma \wedge \overline{\text{TREEFORMULA}}(k, T)$, with the help of interpolation, as described below. The resulting approximations are sound, i.e., preserve the correctness of the resulting strategy.

The PARTITION function (Algorithm 11) computes a local strategy in the root of an abstract game tree. It takes a tuple (T, k, σ), such that T is a certificate tree of height k for $\sigma \in 2^{\mathcal{W}}$ and partitions σ into subsets σ_i such that the controller can win a game of k rounds in the states and against the environment actions contained in σ_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 σ_i where the controller wins by following the branch T_i, and removes σ_i from the current $\hat{\sigma}$. At the next iteration it considers the leftover tree $\tilde{T}$ and state-action set $\hat{\sigma}$.

The algorithm maintains the invariant that $\hat{T}$ is a certificate tree of height k for $\hat{\sigma}$, and hence $\hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \hat{T})$ is unsatisfiable. We decompose this formula into two conjuncts $A \wedge B$ such that A and B only share state and action variables $\mathcal{W}_T$ 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

Algorithm 11 Partitioning winning states

```
1:  function PARTITION(T, k, σ)
2:      σ̂ ← σ
3:      T̂ ← T
4:      for i = 1 to j do
5:          (T_i, T̃) ← SPLIT(T̂)
6:          A ← σ ∧ ¬TREEFORMULA(k, T̃)
7:          B ← TREEFORMULA(k, T_i)
8:          I ← INTERPOLATE(A, B)
9:          σ_i ← I(W_T) ∧ σ
10:         σ̂ ← σ̂ ∧ ¬σ_i
11:         T̂ ← T̃
12:     end for
13:     return [(T_1, σ_1), ..., (T_j, σ_j)]
14: end function
```

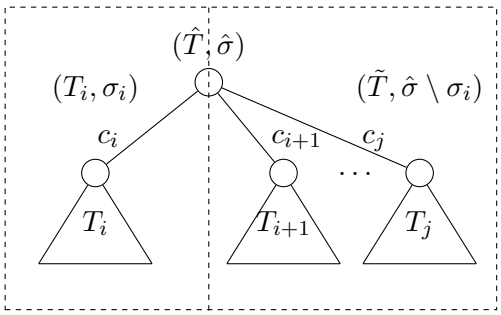


Figure 5.4: Splitting of T in the PARTITION function.

by following the T_i subtree. Hence $\mathcal{I}(\mathcal{W}_T)$ gives us the desired set σ_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 $\hat{\sigma}$ 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 and environment actions in $\hat{\sigma}$ for which the controller cannot win by picking moves from $\tilde{T}$ as a subset. Equivalently, for any state and action in $\hat{\sigma} \wedge \neg\mathcal{I}(\mathcal{W}_T)$, the controller *can* win by following $\tilde{T}$, i.e., $\tilde{T}$ is a certificate tree

for $\hat{\sigma} \wedge \neg\mathcal{I}(\mathcal{W}_T)$, 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 $\hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \hat{T})$.

Proposition 2. $A \wedge B \implies \hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \hat{T})$.

Proof.

$$A \wedge B = (\hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \tilde{T})) \wedge \overline{\text{TREEFORMULA}}(k, T_i)$$

$$A \wedge B = \hat{\sigma} \wedge \Bigg(E(\mathcal{S}_{\tilde{T}}) \vee \bigvee_{\langle e,n\rangle \in \text{SUCC}(\tilde{T})} \delta(\mathcal{S}_{\tilde{T}}, \mathcal{U}_{\tilde{T}}, \mathcal{C}_{\tilde{T}}, \mathcal{S}_n) \wedge \text{ACTION}(e) \wedge \overline{\text{TREEFORMULA}}(k, n) \Bigg)$$
$$\wedge \Bigg(E(\mathcal{S}_{T_i}) \vee (\delta(\mathcal{S}_{T_i}, \mathcal{U}_{T_i}, \mathcal{C}_{T_i}, \mathcal{S}_{n_i}) \wedge \text{ACTION}(e_i) \wedge \overline{\text{TREEFORMULA}}(k, n_i)) \Bigg)$$

$$\implies \hat{\sigma} \wedge \Bigg(E(\mathcal{S}_{\hat{T}}) \vee \bigvee_{\langle e,n\rangle \in \text{SUCC}(\hat{T})} \delta(\mathcal{S}_{\hat{T}}, \mathcal{U}_{\hat{T}}, \mathcal{C}_{\hat{T}}, \mathcal{S}_n) \wedge \text{ACTION}(e) \wedge \overline{\text{TREEFORMULA}}(k, n) \Bigg)$$
$$= \hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \hat{T})$$

□

Proposition 3. *The following invariant is maintained throughout the execution of* PARTITION*: $\hat{T}$ is a certificate tree of height k for $\hat{\sigma}$.*

Proof. We prove by induction. It is a precondition of the function that T is a certificate tree for σ, thus the invariant holds for the initial values $\hat{T} = T$ and $\hat{\sigma} = \sigma$. By the induction hypothesis $(\hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \hat{T}))$ is unsatisfiable, so by Proposition 2 $(A \wedge B)$ must also be unsatisfiable. Hence the interpolation operation in line 8 is well defined. By the properties of interpolants, $(A \implies \mathcal{I})$, hence $(\neg\mathcal{I} \implies \neg A)$ or equivalently $(\neg\mathcal{I} \implies \neg(\hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \tilde{T}))$.

After $\hat{T}$ and $\hat{\sigma}$ are updated in line 11, their new values $\hat{T}'$ and $\hat{\sigma}'$ satisfy the following equalities:

$$\begin{aligned}
\hat{\sigma}' \wedge \overline{\text{TREEFORMULA}}(k, \hat{T}') &= \hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \tilde{T}) \wedge \neg\mathcal{I} \\
&= \neg\mathcal{I} \wedge \hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \tilde{T}) \\
&\implies \neg(\hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \tilde{T})) \\
&\qquad \wedge \hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \tilde{T}) \\
&= \bot
\end{aligned}$$

and hence the invariant is maintained. □

Proposition 4. *Let T be a certificate tree for σ and let $\sigma \wedge \neg E(\mathcal{S}_T) = \bot$. Then $[(T_1, \sigma_1), \ldots, (T_j, \sigma_j)] = \text{PARTITION}(T, k, \sigma)$ is a local winning strategy in the root of T, i.e., the following properties hold:*

1. *$\sigma_1, \ldots, \sigma_j$ is a partitioning of σ:*

$$\sigma = \bigvee \sigma_i \text{ and } \forall i, k.(i \neq k) \implies (\sigma_i \wedge \sigma_k = \bot).$$

2. *T_i is a certificate tree of height k for σ_i.*

Proof. At every iteration of the algorithm, we partition $\hat{\sigma}$ into $\sigma_i = \mathcal{I}(\mathcal{W}_T) \wedge \hat{\sigma}$ and $\hat{\sigma} \wedge \neg\mathcal{I}(\mathcal{W}_T)$. Hence, by construction, no σ_i overlaps with any σ_k.

At the final iteration of the algorithm, the tree $\tilde{T}$ consists of a single root node without outgoing branches. Hence, $A = \hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, \tilde{T}) = \hat{\sigma} \wedge \neg E(\mathcal{S}_{\tilde{T}}) = \hat{\sigma}$. Since $(A \implies \mathcal{I})$, we get $(\hat{\sigma} \implies \mathcal{I})$ and therefore $\mathcal{I} \wedge \hat{\sigma} = \hat{\sigma}$, i.e., all states and actions in $\hat{\sigma}$ are included in the final set σ_j and hence the partitioning completely covers the set σ: $\sigma = \bigvee \sigma_i$.

We prove the second statement of the proposition. The set σ_i is computed as $\mathcal{I}(\mathcal{W}_T) \wedge \hat{\sigma}$ at the ith iteration of the algorithm (line 9). Thus,

$$\sigma_i \wedge \overline{\text{TREEFORMULA}}(k, T_i) = \mathcal{I}(\mathcal{W}_T) \wedge \hat{\sigma} \wedge \overline{\text{TREEFORMULA}}(k, T_i)$$

By the properties of interpolants,

$$\mathcal{I} \wedge B = \mathcal{I}(\mathcal{W}_T) \wedge \overline{\text{TREEFORMULA}}(k, T_i) = \bot.$$

Hence $\sigma_i \wedge \overline{\text{TREEFORMULA}}(k, T_i) = \bot$, i.e. T_i is a certificate tree for σ_i. □

5.1.3 Computing successor states

The NEXT function (Algorithm 12) takes a set σ and its certificate tree T of height k, 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 σ' of the c-successors of σ, such that σ' is winning for the controller and T' is a certificate tree of height $k-1$ for σ'.

Algorithm 12 Successor set

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

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

Proposition 5. *Let T be a certificate tree for σ with a single outgoing edge, labelled c in its root node, and let $(T', \mathcal{I}) =$ NEXT(T, σ). Then:*

1. *$\mathcal{I}$ is an overapproximation of the c-successors of σ, i.e., $\mathcal{I} \supseteq \sigma'$ where*

$$\sigma' = \exists \mathcal{S} \exists \mathcal{U} \exists \mathcal{C} (\delta(\mathcal{S}, \mathcal{U}, \mathcal{C}, \mathcal{S}') \wedge \sigma \wedge c)$$

2. *T' is a certificate tree of length $k-1$ for σ'*

Proof. The c-successor set σ' of σ is defined by $\exists \mathcal{S} \exists \mathcal{U} (\delta(\mathcal{S}, \mathcal{U}, \mathcal{C}, \mathcal{S}') \wedge \sigma \wedge c)$. The matrix of this formula is exactly formula A. Hence the successor set is given by $\sigma' = \exists \mathcal{S} \exists \mathcal{U}(A)$. Since $(A \implies \mathcal{I})$, $\sigma' \implies \exists \mathcal{S} \exists \mathcal{U}(\mathcal{I})$. Since $\mathcal{I}$ is defined over state variables in the root of T' only, the quantifiers can be removed: $\sigma' \implies \mathcal{I}$ or, in the relational form, $\mathcal{I} \supseteq \sigma'$.

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

Algorithm 13 Compiling the winning strategy

```
1: function CompileStrat(Strat)
2:     π ← ⊥,  W ← ⊥
3:     Strat' ← sort(Strat)                    ▷ Sort by descending k
4:     for (w, c, k) ∈ Strat' do
5:         π ← π ∨ (w ∧ ¬W ∧ c)
6:         W ← W ∨ w
7:     end for
8:     return π
9: end function
```

The computed set is an overapproximation of the desired set of successor states but the second property of Proposition 5 guarantees that we can continue the algorithm using the approximation. The interpolant contains additional states that may not be c-successors of σ but we ensure that T' contains a winning strategy for these states. As a result, the partial strategy potentially covers additional unreachable states but is guaranteed to be correct.

5.1.4 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 13) achieves this 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.

Theorem 2 (Correctness of the algorithm)**.** *Let abstract game tree T of height κ be a certificate tree for the set $\sigma = s_0 \wedge \top$, let π be a partial function returned by the strategy generation algorithm, $\pi =$ GENSTRATEGY(T, σ), and let π' be an arbitrary extension of π to a complete function. Then π' is a winning controller strategy for the bounded game of length κ with initial state s_0.*

Proof. GENSTRATEGY generates a strategy π from the set of tuples $Strat$ returned by GENLOCALSTRATS. Each tuple (w, c, k) in $Strat$ is generated by PARTITION and, according to Proposition 4, c is a winning controller move

from w for a game of length k. By Proposition 5, all possible c-successors of w are covered by subsequent iterations of PARTITION and therefore has some tuple $(w', c', k-1)$ in $Strat$. By constructing $\pi = \text{GENSTRATEGY}$ from $Strat$ in the order of highest k to lowest, we ensure that any mapping defined by π for w' is from a tuple with height $\geq k-1$ and so has a strategy that is winning for the controller for a game of $k-1$ rounds or more.

Therefore π defines a controller action for every state reachable from s_0 by playing the strategy. Any extension of π to a complete function that maps unreachable states to an arbitrary controller action must then be a winning strategy since the controller is guaranteed to stay within the safe region W by playing π. □

5.2 Optimisations

5.2.1 Strategy extraction with learning

In Section 4.2.1 I described an optimisation to bounded realisability in which computational learning is used to exclude known losing states from the search tree. If this optimisation is enabled then the final certificate tree returned by the realisability algorithm will assume that these states are winning. As a result the tree will not contain the controller actions required to win from these states. In this case we need to adjust the way that strategy extraction is done so that the generated strategy includes these actions.

The modification is simple: partial strategies are extracted online during the realisability check. Recall that a set of states b is learned to be losing for the environment for height k when $b \wedge \overline{\text{TREEFORMULA}}(k, T)$ is unsatisfiable for some subtree T. In other words, T is a certificate tree for b. Learning occurs in FINDCANDIDATE (Algorithm 4), which calls LEARN (Algorithm 8) whenever an unsatisfiable formula is generated. All subsequent calls to FINDCANDIDATE use a modified version of $\overline{\text{TREEFORMULA}}$ that assumes that the environment cannot win from (b, k). Algorithm 14 reproduces LEARN with an additional call to GENLOCALSTRATS in line 12. Instead of invoking GENLOCALSTRATS once on the final certificate tree we invoke GENLOCALSTRATS on the certificate tree for every learned state. We collect each of these partial strategies in a set $Strats$, which is combined into a winning partial strategy with COMPILESTRAT after bounded realisability terminates.

Algorithm 14 Learning with online strategy extraction

```
function LEARN(p, s, k, T, f)
    ŝ ← s
    for s ∈ S do
        sol ← SATWITHASSUMPTIONS(ŝ \ {s}, f)
        if sol = NULL then
            ŝ ← ŝ \ {s}
        end if
    end for
    if p = cont then
        B^c ← B^c ∨ ŝ
    else
        Strats ← Strats ∪ GENLOCALSTRATS(T, k, ŝ ∧ ⊤)
        for i ∈ [0, ..., k] do
            B^e[i] ← B^e[i] ∨ ŝ
        end for
    end if
end function
```

Theorem 3. *Let G be a bounded safety game that is realisable with respect to an error set E and a bound k. Let $Strats$ be a set of partial strategies generated during* SOLVEABSTRACT, *then π = COMPILESTRAT($Strats$) is a winning partial controller strategy for G.*

Proof. We prove correctness inductively. At the first online call to GENLOCALSTRATS, B^e is empty and by Theorem 2 $Strat_0$ is a partial strategy for a state-round pair (b_0, k_0). $Strat_0$ is added to $Strats$.

In subsequent calls all interpolants are constructed such that for every $(b, k) \in B^e$, b is assumed to be winning for the controller at any height $< k$. So, $Strat_{j+1}$ is a partial strategy for a game beginning at b_{j+1} and of length k_{j+1} where every state-round pair $(b_0, k_0), \ldots, (b_k, k_j)$ is assumed to be winning. As in Theorem 2, all tuples (w, c, k) in $Strat_{j+1}$ are such that c is a winning controller action from w for a game of length k if all $(b_0, k_0), \ldots, (b_k, k_j)$ is assumed to be winning. Likewise, every c-successor of w is either covered by $Strat_{j+1}$ or is contained in $(b_0, k_0), \ldots, (b_k, k_j)$.

Under the inductive hypothesis, $Strats = Strat_0 \cup \ldots \cup Strat_j$ contains winning partial strategies for every $(b_0, k_0), \ldots, (b_j, k_j)$. Thus every state assumed to be winning at the $j + 1$ iteration has a winning partial strategy in $Strat_0 \cup \ldots \cup Strat_j$ and so $Strat_0 \cup \ldots \cup Strat_{j+1}$ contains winning partial

strategies for all $(b_0, k_0), \ldots, (b_{j+1}, k_{j+1})$.

Thus, $Strats$ contains winning partial strategies for every state in B^e. SOLVEABSTRACT terminates when it discovers a certificate tree for the initial state s_0. As with all certificate trees generated during execution the algorithm learns from the tree via a call to LEARN. Thus, at termination $s_0 \in B^e[k]$ and $Strats$ contains a winning partial strategy from s_0. Therefore $\pi = \text{COMPILESTRAT}(Strats)$ is a winning partial controller strategy for the game. □

5.2.2 Ensuring compact interpolants

The implementation of strategy extraction (described in further detail in Chapter 7) also contains an optimisation that ensures that the size of the formulas used in the algorithm do not grow too large. Strategy extraction generates many CNF formulas that are used to construct interpolants. Those formulas also contain interpolants themselves, which are arbitrary formulas that must first be converted to CNF. The usual Tseitin transformation converts a formula to CNF by introducing variables and potentially increases the time taken to determine (un)satisfiability and construct a new interpolant. We optimise the implementation by constructing a BDD from each interpolant and using the BDD to efficiently enumerate cubes. A set of cubes usually requires fewer additional variables than an arbitrary circuit to represent in CNF.

BDD sweeping is used to reduce the size of formulas by carefully constructing BDDs from subformulas to deduplicate equivalent parts of the formula [Kuehlmann and Krohm, 1997]. The sweep is done carefully so that no one BDD grows too large. For this optimisation a complete BDD sweeping implementation was not required since in practice the BDDs constructed from these interpolants never grew too large. However, if a problem instance was found to generate interpolants that are difficult to represent then BDD sweeping could be used here.

5.3 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]. 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, 2013].

5.4 Summary

I have presented a strategy extraction algorithm to extend bounded realisability to bounded synthesis. The performance overhead of this approach will be evaluated in Chapter 7.

- A strategy may be extracted from the certificate tree generated by the bounded realisability algorithm. The winning controller actions are given in the edges of the certificate tree and the algorithm must simply assign actions to subsets of the combined state and environment action space.

- A set of states and environment actions may be partitioned such that a controller action is winning for each partition. This can be approximated efficiently using Craig interpolation. Likewise, for a set of state and environment actions the successor states with respect to a controller action may be approximated by an interpolant. These two operations implemented by efficient approximations are sufficient to construct a strategy from a certificate tree.

- The algorithm may also be used, with some modification, to extract a spoiling strategy for the environment. Another modification enables

compatibility with the computational learning optimisation of bounded realisability.

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. In model checking, a more nuanced approach is to use the diameter of the system [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. An analogous approach for solving games would require computing the longest path the environment can enforce in the game and setting the bound to its length. Computing a sufficient bound is expensive and the subsequent bounded realisability check may be infeasible if the bound is high.

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. In Section 4.2.1 I presented an optimisation that learns a subset of states losing for one of the players every time a candidate cannot be found for an abstract game. In this chapter I present I present an improved learning procedure that maintains certain properties for sets of learned states. The bounded algorithm is reproduced in Algorithm 15 with additional calls to the new learning procedure. 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.

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. When a player cannot win a abstract game of k rounds from a state s we have proven that s is a losing state for any game with a height of k. We can learn additional losing states from the tree via interpolation. This is achieved in line 19 in Algorithm 15, enabled in the unbounded version of the algorithm, which invokes LEARN or $\overline{\text{LEARN}}$ to learn controller or environment losing states respectively (Algorithm 16).

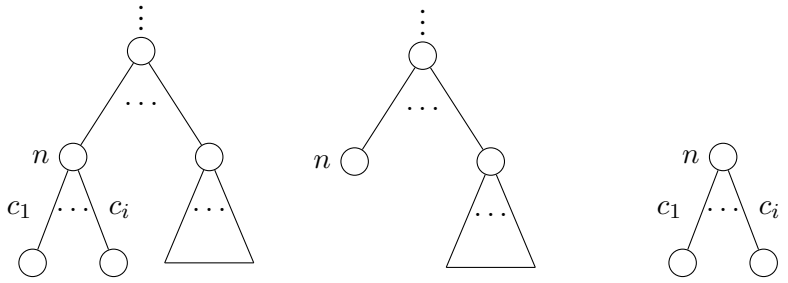
6.1.1 Learning with interpolation

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.

We use these properties to learn the states that are losing to the actions in every subgame in an abstract game tree T beginning in a state s. We assume that T is labelled with controller actions and that s is losing for the environment; learning controller losing states is described below. We choose a non-leaf node n of T with maximal depth, i.e., a node whose children are leafs (Algorithm 16, line 3), with the aim of learning the states that lose to the subgame beginning at n. Figure 6.1a shows a fragment

Algorithm 15 Unbounded realisability

function SOLVEABSTRACT(p, s, k, T)
  $cand \leftarrow$ FINDCANDIDATE(p, s, k, T)
  if $k = 1$ **then return** $cand$
  $T' \leftarrow T$
  loop
    if $cand =$ `NULL` **then return** `NULL`
    $\langle cex, l, u\rangle \leftarrow$ VERIFY($p, s, k, T, cand$)
    if $cex =$ `false` **then return** $cand$
    $T' \leftarrow$ APPEND(T', l, u)
    $cand \leftarrow$ SOLVEABSTRACT(p, s, k, T')
  end loop
end function
function FINDCANDIDATE(p, s, k, T)
  $\hat{T} \leftarrow$ EXTEND(T)
  $f \leftarrow$ **if** $p =$ `cont` **then** TREEFORMULA($k, \hat{T}$) **else** $\overline{\text{TREEFORMULA}}(k, \hat{T})$
  $sol \leftarrow$ SAT($s(\mathcal{S}_{\hat{T}}) \wedge f$)
  if $sol =$ `unsat` **then**
    ▷ Unbounded solver learns states here
    if $p =$ `cont` **then** LEARN($s, \hat{T}$) **else** $\overline{\text{LEARN}}(s, \hat{T})$
    return `NULL`
  else
    return $\{\langle n, c\rangle | n \in$ NODES(T) $, c =$ SOL(n)$\}$
  end if
end function
function VERIFY($p, s, k, T, cand$)
  for $l \in$ LEAVES(T) **do**
    $\langle k', s'\rangle \leftarrow$ OUTCOME($s, k, cand, l$)
    if $p =$ CONT **then**
      $T' \leftarrow \emptyset$
    else
      $T' \leftarrow \{cand(l)\}$
    end if
    $a \leftarrow$ SOLVEABSTRACT(OPPONENT(p), s', k', T')
    if $a \neq$ `NULL` **then return** $\langle$`true`$, l, a\rangle$
  end for
  return $\langle$`false`$, \emptyset, \emptyset\rangle$
end function



(a) T (b) T_1 (c) T_2

Figure 6.1: Splitting a certificate tree

of T containing n. First, we then split the tree at n such that both slices T_1 and T_2 contain a copy of n (line 4). Figure 6.1b shows T_1, which contains all of T except the children of n, and T_2 (Figure 6.1c), which contains only n and its children. There is no candidate strategy for T so the formula constructed by unrolling copies of transition relation with fixed controller actions, $s \wedge \overline{\text{TREEFORMULA}}(k, T)$, 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 6.1c. 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.

We have discovered a set $\mathcal{I}$ of states losing for the environment and now must record it. Care must be taken in this step because the learning procedure operates on a bounded game. We have learned that there is no environment strategy that forces the game into an error state from $\mathcal{I}$ in k rounds or less but there may be a longer strategy that does force an error. 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 $\top$ for all $k > 0$ and $B^m[0] \leftarrow E$. Whenever an environment-losing set $\mathcal{I}$ is discovered for a node n with bound $\text{HEIGHT}(k, n)$ in line 13 of Algorithm 16, this set is subtracted

Algorithm 16 Learning algorithms

Require: $s(\mathcal{S}_T) \wedge \textsc{TreeFormula}(k, T) \equiv \bot$
Require: *Must-invariant* holds
Ensure: *Must-invariant* holds
Ensure: $s \wedge B^M \not\equiv \bot$ ▷ s will be added to B^M

function LEARN(s, T)
  if SUCC(T) $= \emptyset$ **then return**
  $n \leftarrow$ non-leaf node with min height
  $\langle T_1, T_2 \rangle \leftarrow$ GTSPLIT(T, n)
  $F_1 \leftarrow s(\mathcal{S}_T) \wedge \textsc{TreeFormula}(k, T_1)$
  $F_2 \leftarrow \textsc{TreeFormula}(k, T_2)$
  $\mathcal{I} \leftarrow$ INTERPOLATE(F_1, F_2)
  $B^M \leftarrow B^M \vee \mathcal{I}$
  LEARN(s, T_1)
end function

Require: $s(\mathcal{S}_T) \wedge \overline{\textsc{TreeFormula}}(k, T) \equiv \bot$
Require: *May-invariant* holds
Ensure: *May-invariant* holds
Ensure: $s \wedge B^m[\textsc{Height}(k, T)] \equiv \bot$ ▷ s will be removed from B^m

function $\overline{\text{LEARN}}$(s, T)
  if SUCC(T) $= \emptyset$ **then return**
  $n \leftarrow$ non-leaf node with min height
  $\langle T_1, T_2 \rangle \leftarrow$ GTSPLIT(T, n)
  $F_1 \leftarrow s(\mathcal{S}_T) \wedge \overline{\textsc{TreeFormula}}(k, T_1)$
  $F_2 \leftarrow \overline{\textsc{TreeFormula}}(k, T_2)$
  $\mathcal{I} \leftarrow$ INTERPOLATE(F_1, F_2)
  if strategy generation enabled **then**
    for $i = 1$ to HEIGHT(k, n) **do**
      $Strats[i] \leftarrow Strats[i] \cup \textsc{GenLocalStrats}(\mathcal{I}, T_2), Strats[i]$
    end for
  end if
  for $i = 1$ to HEIGHT(k, n) **do**
    $B^m[i] \leftarrow B^m[i] \setminus \mathcal{I}$
  end for
  $\overline{\text{LEARN}}(s, T_1)$
end function

from $B^m[i]$, for all i less than or equal to the bound (lines 14–16).

Learning of states losing for the controller is similar (LEARN in Algorithm 16). The main difference is that environment-losing states are losing for all game heights. Therefore we record these states in a single set B^M of must-losing states (Algorithm 16, line 6). This set is initialised to the error set E and grows as new losing states are discovered.

Once the set of states losing for the subtree T_2 has been recorded we continue to learn states from the other nodes in T_1. This achieved by recursively splitting the already reduced T_1 and learning the states that lose to a different subtree. In order to compute interpolants we have to ensure that the formulas constructed from trees during recursive calls are unsatisfiable. For controller losing trees we may have removed a subtree in which the environment forces a visit to an error state and enabled a satisfying assignment to the SAT query. To handle this case we introduce constraints to TREEFORMULA (and similarly $\overline{\text{TREEFORMULA}}$) to take into account previously learned states. Intuitively, we capture T_2 by forbidding the player from visiting the states that lose to the actions in the tree. As a result any pairs of formulas constructed during the recursive decomposition of T are unsatisfiable and enable interpolation.

6.1.2 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 6.2 shows the variables (6.4a), the initial state of the system (6.4b), and the formulas for computing next-state variable assignments (6.4c) for this example. We use primed identifiers to denote next-state variables and curly braces to define the domain of a variable.

Consider node n in Figure 6.3a, 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. Specifically,

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 6.2: Example

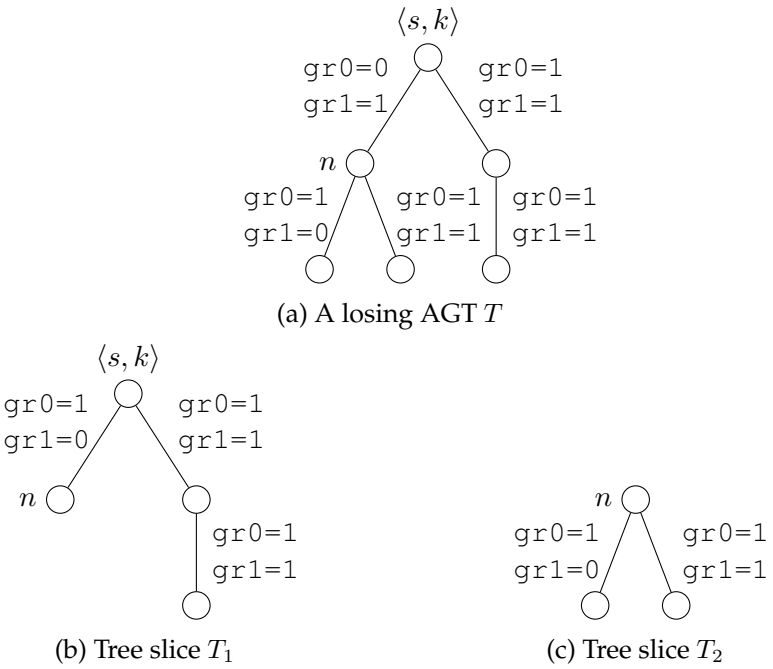
we compute a subset of such states, using interpolation, which we will show is sufficient to ensure convergence.

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 6.3c. 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.

6.1.3 Convergence on a fixed point

Algorithm 18 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 now prove that the algorithm will eventually converge on a fixed point and that it guarantees that the game is realisable.



(a) A losing AGT T

(b) Tree slice T_1

(c) Tree slice T_2

Figure 6.3: Splitting of an abstract game tree by the learning procedure.

We define two global invariants of the algorithm. The *may-invariant* states that sets $B^m[i]$ decrease monotonically with i and that each $B^m[i+1]$ overapproximates the states from which the environment can force the game into $B^m[i]$, i.e. $B^m[i+1]$ overapproximates the uncontrollable predecessor of $B^m[i]$. The *must-invariant* guarantees that the must-losing set B^M is an underapproximation of the actual losing set B. Both invariants trivially hold after B^m and B^M have been initialised in the beginning of the algorithm. Further interaction of the algorithm with the recorded learned states is constrained to LEARN and $\overline{\text{LEARN}}$ so it suffices to prove these invariants for these functions only.

Proposition 6. *Assuming the preconditions are met,* $\overline{\text{LEARN}}$ *satisfies its postconditions. Namely,*

1. $\overline{\text{LEARN}}$ *maintains the may-invariant:*

$$\forall(0 < i < k).\ B^m[i] \subseteq B^m[i+1], Upre(B^m[i]) \subseteq B^m[i+1].$$

2. $\overline{\text{LEARN}}$ *ensures that* s *is removed from* B^m*:*

$$s \wedge B^m[\text{HEIGHT}(k, T)] \equiv \bot$$

Algorithm 17 Tree formula construction with B^m and B^M

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:

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

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

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

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

Proof. The precondition of $\overline{\text{LEARN}}$ states that $s(\mathcal{S}_T) \wedge \overline{\text{TREEFORMULA}}(k, T) \equiv \bot$. We use the unsatisfiability of this formula to construct an interpolant. 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 by the precondition. 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

Algorithm 18 Unbounded Synthesis

```
1: function SOLVEUNBOUNDED(G, E)
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] ← ⊤
14:        CHECKBOUND(k)
15:    end for
16: end function
Require: May and must invariants hold
Ensure: May and must invariants hold
Ensure: s_0 ∉ B^m[k] if there exists a winning controller strategy with bound k
Ensure: s_0 ∈ 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
```

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.

□

Proposition 7. *Assuming the preconditions are met,* LEARN *satisfies its postconditions. Namely,*

1. LEARN *maintains the* must-invariant*:*

$$B^M \subseteq B.$$

2. LEARN *ensures that* s *is added to* B^M*:*

$$s \wedge B^M \not\equiv$$

Proof. The proof of LEARN is similar to the above proof of $\overline{\text{LEARN}}$. 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 $s \wedge B^M \not\equiv \bot$, the second postcondition, is true. □

Proposition 8. *We prove the following for* CHECKBOUND

1. CHECKBOUND *maintains the may and must invariants*

2. *If the bounded game is realisable for* k *then* CHECKBOUND *terminates and guarantees* $s_0 \notin B^m[k]$.

3. *If the bounded game is unrealisable for* k *then* CHECKBOUND *terminates and guarantees* $s_0 \in B^M$.

Proof. B^m and B^M are updated inside $\overline{\text{LEARN}}$ and LEARN only. Therefore, by Propositions 6 and 7, CHECKBOUND maintains both invariants.

From the correctness of SOLVEABSTRACT given in Theorem 1 we ensure that CHECKBOUND terminates and during execution a certificate tree is generated and checked by FINDCANDIDATE. In the modified version of SOLVEABSTRACT, one of the learning procedures is called when FINDCANDIDATE returns `NULL`. Thus CHECKBOUND generates a call to a learning procedure and passes s and a certificate tree T. Therefore, when the game is realisable,

Proposition 6 guarantees that s is removed from $B^m[k]$. Likewise, when the game is unrealisable Proposition 7 guarantees that s is removed from B^M.

□

Theorem 4. *Let G be a game with a safety condition defined by a set of error states E.* SOLVEUNBOUNDED(G, E) *is guaranteed to terminate and correctly decide realisability for G.*

Proof. Assume that G is realisable, we use the may-invariant to show correctness of SOLVEUNBOUNDED. The invariant guarantees that $B^m[i]$ contains all states from which the environment can force the game into an error state in at most i steps. If $B^m[i] \equiv B^m[i+1]$ (line 6) then the invariant states that $Upre(B^m[i]) \subseteq B^m[i]$. Thus, any state that the environment can use to force the game into $B^m[i]$ is contained within $B^m[i]$. In other words, $B^m[i]$ over-approximates the winning states for the environment. Proposition 8 ensures that $s_0 \notin B^m[k]$, and since the invariant ensures 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.

Alternatively, G is unrealisable. 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`.

Termination also follows from the invariants. Given that B^m decreases monotonically and there are a finite set of states in the game, the algorithm is guaranteed to reach a fixed point if the game is realisable. If the game is unrealisable there must exist a finite length winning environment strategy, thus by iteratively increasing k we guarantee termination.

□

6.1.4 Strategy extraction

In the previous chapter I showed how to extract a strategy from the certificate tree generated during the bounded realisability algorithm. I will now prove that a similar approach extracts a correct strategy for an unbounded game when strategy generation is done online as it is when the learning optimisation is enabled for bounded realisability. In order to extract strategies online, line 20 becomes active in $\overline{\text{LEARN}}$ and a local strategy is generated for each learned interpolant. The local strategies are collected in an array $Strats$, which is

Algorithm 19 Compiling the winning strategy

```
1: function CompileStrat(Strats, i)
2:     π ← ⊥,  W ← ⊥
3:     for (w, c, k) ∈ Strats[i] do
4:         π ← π ∨ (w ∧ ¬W ∧ c)
5:         W ← W ∨ w
6:     end for
7:     return π
8: end function
```

indexed by the height of the subtree that enabled interpolation. Algorithm 19 shows a modified version of CompileStrat that takes this array and the index of B^m such that $B^m[i+1] = B^m[i]$, and returns a winning strategy for the game. As proved below, the local strategies from the sets of states removed from the fixed point of B^m are sufficient to construct a winning strategy for the controller.

Theorem 5. *Let G be a safety game that is realisable with respect to an error set E. Let $Strats$ be a set of partial strategies generated during* SolveUnbounded(G, E)*, then $\pi =$* CompileStrat$(Strats)$ *is a winning partial controller strategy for G.*

Proof. Let (b, k) be a pair such that b is a set of states learned to be environment-losing for a game of length k during execution of $\overline{\text{LEARN}}$. Then (b, k) has corresponding tree, T, that has a depth of one. We construct $[(w_0, c_0, k_0), \ldots, (w_j, c_j, k_j)] =$ genLocalStrats(T, k, b). Since the tree is of depth one, NEXT is never called. From the precondition of $\overline{\text{LEARN}}$ we know that $b(\mathcal{S}_T) \wedge \overline{\text{treeFormula}}(k, T)$ is false. Therefore interpolation in PARTITION is well defined.

On termination of unbounded realisability we have an index i such that $B^m[i+1] = B^m[i]$. Every set of states removed from $B^m[i+1]$ has a corresponding local strategy in $Strats[i+1]$ that consists of one or more tuples (w, c, k). By Proposition 4 (in Chapter 5), $b(\mathcal{W}) = \bigvee_i w_i$, i.e. there is a (w, c, k) tuple for every state and environment action action in b.

Recall that each tuple is constructed from an interpolant that guarantees $w \wedge \overline{\text{treeFormula}}(i+1, T) = \bot$ where T is a tree consisting of one edge

labelled c. We may expand the formula like so:

$$
\begin{aligned}
& w \wedge \overline{\text{TREEFORMULA}}(i+1, T) \\
=& w \wedge B^m[i+1](\mathcal{S}_T) \wedge \Bigg(E(\mathcal{S}_T) \vee \\
& \bigvee_{\langle e,n\rangle \in \text{SUCC}(T)} (\delta(\mathcal{S}_T, \mathcal{U}_n, \mathcal{C}_e, \mathcal{S}_n) \wedge \mathcal{C}_e = \text{ACTION}(e) \wedge \overline{\text{TREEFORMULA}}(i, n)) \Bigg) \\
=& w \wedge B^m[i+1](\mathcal{S}_T) \wedge \Bigg(E(\mathcal{S}_T) \vee \\
& (\delta(\mathcal{S}_T, \mathcal{U}_n, \mathcal{C}_e, \mathcal{S}_n) \wedge \mathcal{C}_e = c \wedge \overline{\text{TREEFORMULA}}(i, n)) \Bigg) \\
=& w \wedge B^m[i+1](\mathcal{S}_T) \wedge \Bigg(E(\mathcal{S}_T) \vee (\delta(\mathcal{S}_T, \mathcal{U}_n, \mathcal{C}_e, \mathcal{S}_n) \wedge \mathcal{C}_e = c \wedge B^m[i](\mathcal{S}_n)) \Bigg)
\end{aligned}
$$

By choosing c in $w \wedge B^m[i+1]$ the controller guarantees that the environment cannot reach any state in the current set $B^m[i]$. Each set in B^m decreases over the execution of the algorithm, so this property holds for the final value of $B^m[i]$. Since $B^m[i+1]$ excludes only states for which this property is also true we may simplify the property to state that from any w the controller can guarantee that the controller cannot reach $B^m[i]$.

Since $B^m[i] = B^m[i+1]$, the only states excluded from $B^m[i]$ are the same as those removed from $B^m[i+1]$ in $\overline{\text{LEARN}}$. So the set of successor states of (w, c) are contained within the set of all pairs $(b, i+1)$. Thus, all states reachable by playing local strategies $Strats[i+1]$ have defined local strategies also in $Strats[i+1]$.

By Proposition 8, $s_0 \notin B^m[i+1]$, so there is a strategy defined for all states and environment actions in the initial state. As shown in Theorem 4, $B^m[i+1]$ overapproximates the winning region of the environment. Therefore, if the controller plays according to $\pi = \text{COMPILESTRAT}(Strats, i+1)$ there is a defined strategy from the initial state that ensures that the game stays outside the environment's winning region. □

6.2 Optimisations

6.2.1 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. Note that it is possible to discover multiple sets in this way by iterating the procedure with different orders on the variables removed from s_0. However, in practice the benefit of these additional sets was found to be outweighed by the cost of computing them.

Algorithm 20 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
```

6.2.2 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 potentially 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 assignment s' is also losing to the actions in T. This increases F_1 and we may learn a larger interpolant. As with the previous optimisation it would be possible to learn multiple assignments but this was not found to be beneficial in practice.

6.2.3 Improving candidate strategies

Using heuristics to strengthen candidates in FINDCANDIDATE is an important avenue for future optimisations as the quality of candidate strategies affects the rate of convergence of the counterexample guided search. Here I introduce some possible heuristics but leave a thorough investigation to future work. The key insight is that the algorithm remains correct as long as FINDCANDIDATE returns a strategy whenever one exists. There is significant freedom to choose strategies according to a heuristic without losing soundness or completeness.

In Chapter 4 I introduced strategy shortening to improve candidate strategies in the bounded realisability algorithm. This is a heuristic that searches for candidates that may reach known winning states earlier in the game. For unbounded synthesis this optimisation 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 potentially be proven to be losing for larger values of k.

Another example heuristic is an additional SAT call in FINDCANDIDATE that first checks 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 controller strategy that forces a loop may be able to force the game to stay within a set of states that are reachable inside the loop. If this is possible then the algorithm can quickly converge on a fixed point using this strategy to keep the game outside the overapproximation of then environment's winning region. In practice, such strategies are rarely found and the additional call to the SAT solver was a waste of resources.

Another heuristic for future 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. For instance, the quantifier prefix on the formula constructed from a tree could be modified to $\forall \mathcal{U}_k \exists \mathcal{C}_k \exists \mathcal{U}_{k-1} \exists \mathcal{C}_{k-1} \ldots \exists \mathcal{U}_1 \exists \mathcal{C}_1$ so that there must exist a winning run for all possible actions the environment may take in the first round of the game. This may help ensure that the controller's candidate strategy does not rely on the environment choosing a particular action in that round and reduce the number of refinements to the game abstraction.

6.3 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 approximation via interpolation, which allows completeness without sacrificing performance to an exploding symbolic representation.

6.3.1 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 ideology to the approach described here but are used to solve a different problem.

6.3.2 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.

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 6.4: Parameterised counter example

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 6.4 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 6.5a. The controller strategy shown in this tree is to not reset the counter. 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

(a) No reset candidate (b) Reset candidate

Figure 6.5: Game trees for the counter specification

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 practice, unbounded synthesis can do well on this specification when the SAT solver gives a candidate such as in Figure 6.5b. 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.

6.3.3 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. The example as described is the $n = 2, m = 2$ instance of the more general problem of an arbiter of n resources and an environment that can make up to m requests at once. The general problem is 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 ${}^nC_m$ 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.

6.4 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

In this chapter I present benchmarking results for the implementations of each previous chapter. Bounded realisability is implemented in a tool called EVASOLVER as joint work between Nina Narodytska and myself. EVASOLVER was built in C++ and is based on the source code of RAREQS [Janota et al., 2012]. The tool calls out to Glucose [Audemard and Simon, 2009] for SAT solving. Strategy extraction was later added to EVASOLVER using PeRIPLO [Rollini et al., 2013] to construct interpolants. The implementation also uses cudd [Somenzi, 2001] to reduce interpolants into cubes via BDDs. I implemented unbounded realisability in a separate open source tool, TERMITESAT [Legg, 2016], which contains a reimplementation of the bounded realisability algorithm. TERMITESAT is written in Haskell and also uses Glucose for SAT solving, PeRIPLO for interpolant construction, and cudd to reduce interpolants to cubes. TERMITESAT was submitted to the 2016 synthesis competition and the results are shown below. As part of that submission I added a hybrid mode to TERMITESAT that runs the unbounded synthesis algorithm in parallel with Simple BDD Solver [Walker, 2014].

7.1 Bounded realisability

The algorithm is evaluated on four families of benchmarks derived from driver synthesis problems. Bounded realisability is only able to prove the absence of counterexample traces up to a certain length for safety games so the tool is instead evaluated on benchmarks formulated as the dual reachability game. EVASOLVER solves games in which the controller must *reach* a goal state. The benchmarks are equivalent to unrealisable safety specifications. Each benchmark models the data path of one 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 [Henzinger et al., 2003]. The transition relation of the game is given in the form of variable update functions $s' := f(\mathcal{S}, \mathcal{U}, \mathcal{C})$, one for each state variable $s \in \mathcal{S}$.

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 [Janota et al., 2012] and DEPQBF [Lonsing and Biere], having first run them through the bloqqer [Biere et al., 2011] preprocessor. Second, we solve our benchmarks using the Termite [Walker and Ryzhyk, 2014] BDD-based solver that uses dynamic variable reordering, variable grouping, transition relation partitioning, and other optimisations.

Our experiments, summarised in Figures 7.1 to 7.4, 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 outperforms EVASOLVER on these benchmarks. However, in the two other benchmarks, Termite does not find a compact BDD-based representation of the winning set. EVASOLVER outperforms 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.

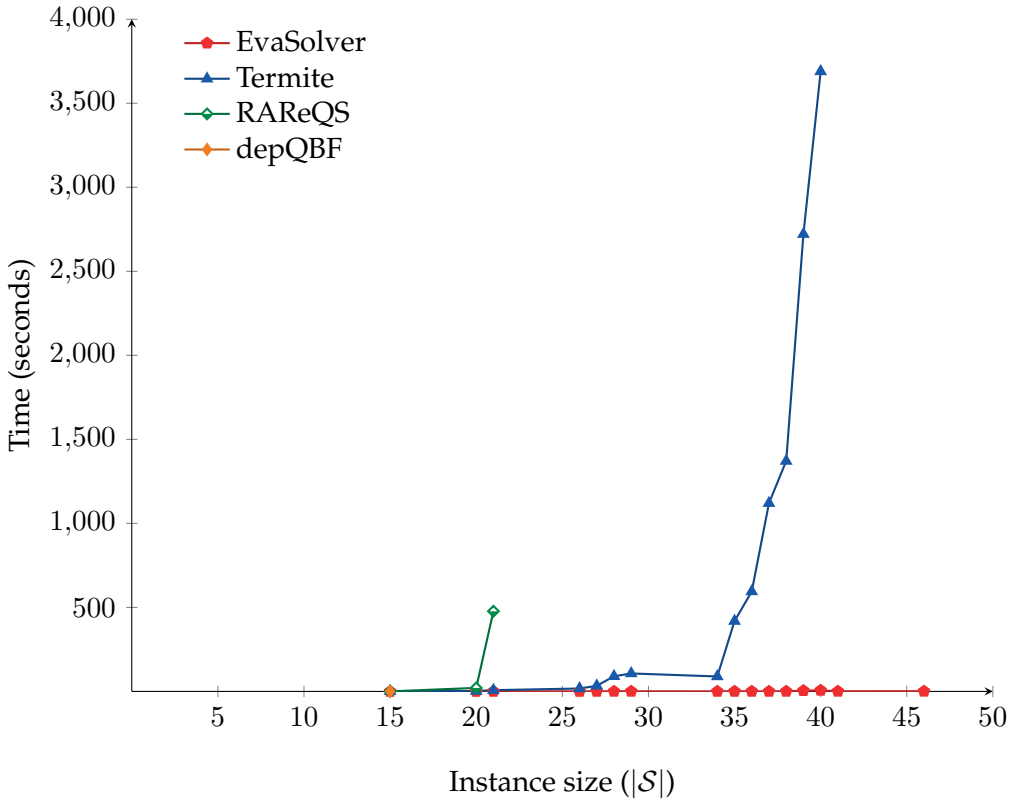


Figure 7.1: UART

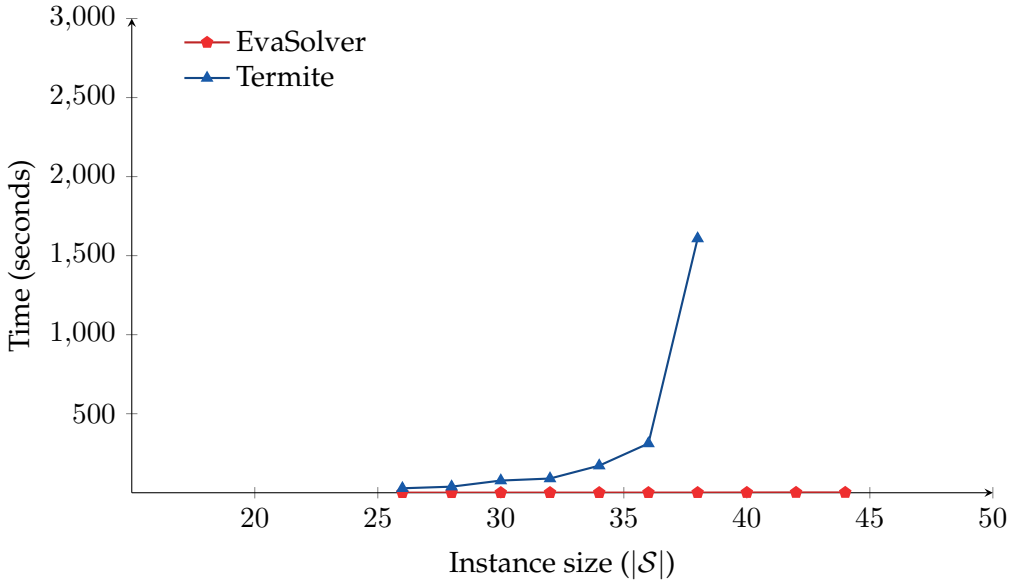


Figure 7.2: IDE DMA

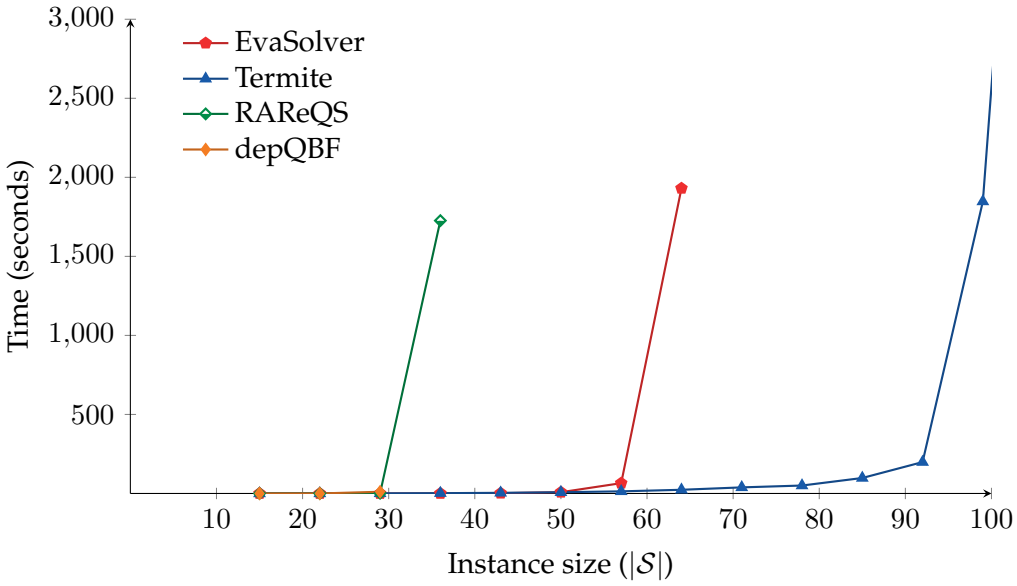


Figure 7.3: SPI

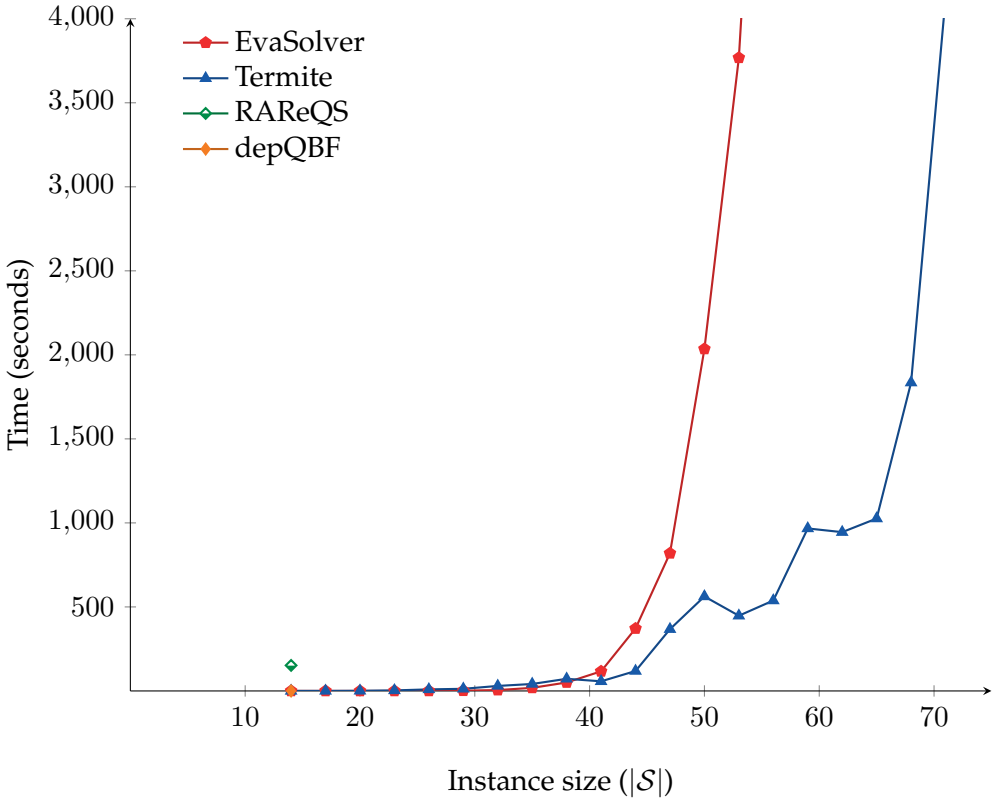


Figure 7.4: Ethernet

7.2 Strategy extraction

Strategy extraction is evaluated on the same set of driver synthesis benchmarks. Figures 7.5 to 7.8 sumarise the results. For each family, we show strategy generation time as a function of the number of state variables in the specification for a selection of the hardest instances of the family solved by EVASOLVER. Specifically, we show the time it took the base realisability solver to determine the winner (the realisability line), as well as the total time taken to solve the game and compute the winning strategy (the total 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** column shows the overhead of strategy extraction.

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, I report the effective overhead (**EffOH**) of strategy extraction if this engineering effort had been done.

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 the 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. The size of an interpolant is reported by the size of its corresponding BDD. This conversion is done by the tool as an optimisation to simplify interpolants into cubes.

These 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.

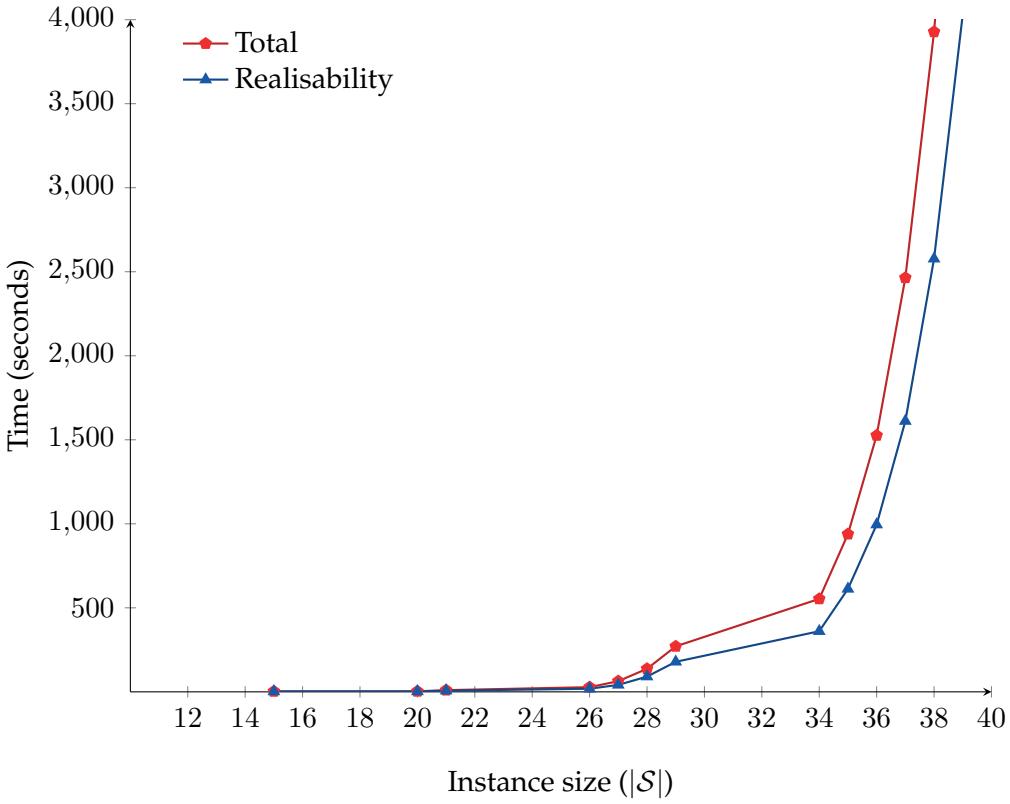


Figure 7.5: Strategy Extraction (UART)

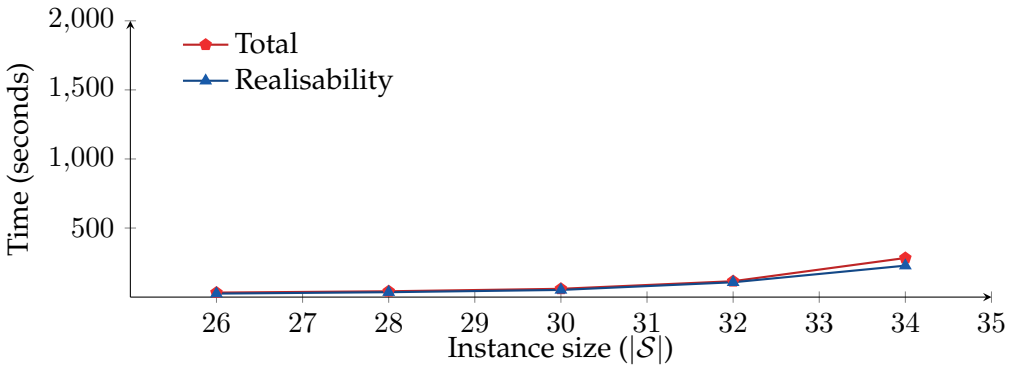


Figure 7.6: Strategy Extraction (IDE DMA)

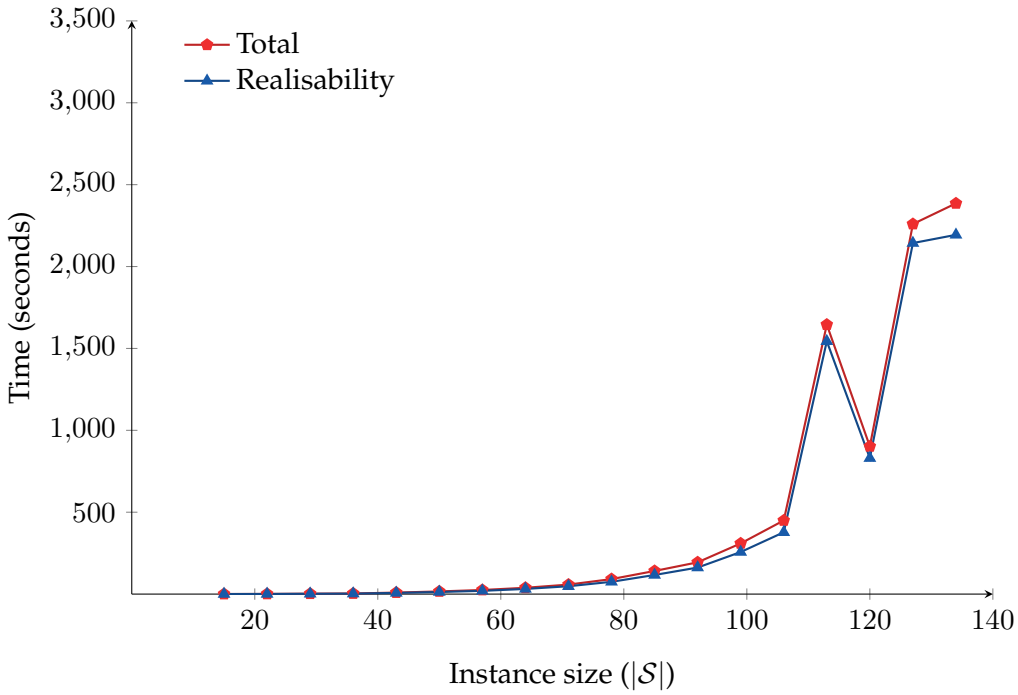


Figure 7.7: Strategy Extraction (SPI)

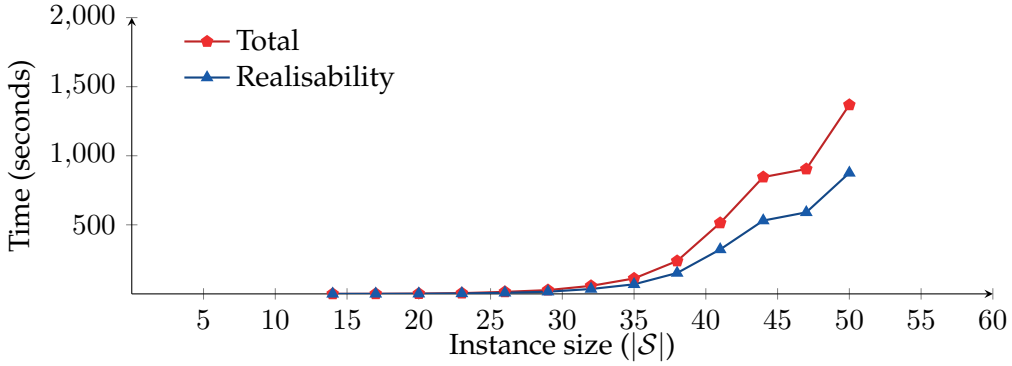


Figure 7.8: Strategy Extraction (Ethernet)

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

Table 7.1: Detailed strategy extraction results

7.3 Unbounded realisability

Unbounded realisability is evaluated on the benchmarks of the 2015 synthesis competition (SYNTCOMP'15). I also show the results of the 2016 competition, which TERMITESAT was submitted to. Each competition 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 [Jacobs et al., 2015].

7.3.1 Benchmarking

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 this benchmarking are shown, along with the synthesis competition results [SYNTCOMP, 2015], 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.

I also present a cactus plot of the number of instances solved over time (Figure 7.9). We have plotted the best configuration of each solver we bench-

marked. The solvers shown are DEMIURGE [Bloem et al., 2014], the only SAT-based tool in the competition, the winner of the sequential realisability track SIMPLE BDD SOLVER 2 [Walker and Ryzhyk, 2014], and AbsSynthe (seq3) [Brenguier et al., 2014].

The results show that although TERMITESAT does not solve as many total instances as other solvers, it is able to solve a significant number of problem instances that no other individual solver could. Although TERMITESAT was the only sequential solver to decide these instances, in the parallel track of the competition DEMIURGE was able to solve many of them by combining several different approaches. A combination of unbounded synthesis, the SAT approaches in DEMIURGE, and a traditional BDD solver may be a significant step forward for synthesis.

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		
TermiteSAT		*103*	*12*

Table 7.2: Synthesis Competition 2015 Results

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

Table 7.3: Instances uniquely solved by our approach

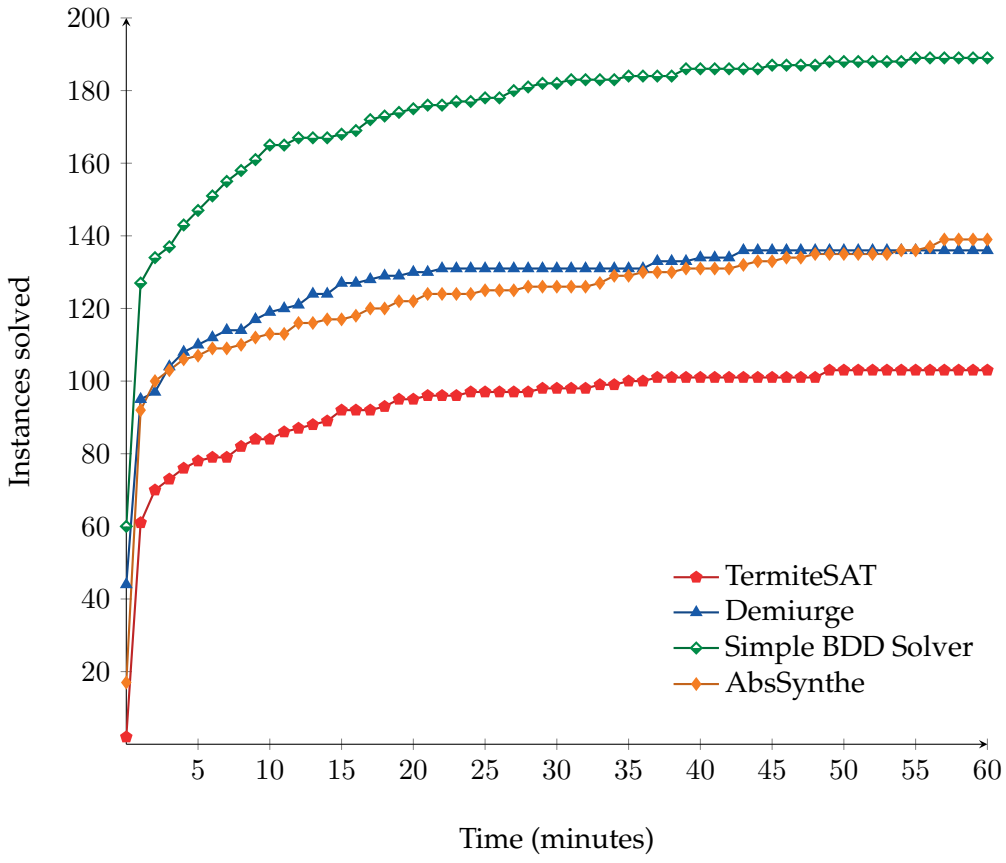


Figure 7.9: Number of instances solved over time.

Solver	Solved	Unique
Simple BDD Solver (w/ Abstraction)	175	1
Simple BDD Solver (w/ Abstraction 2)	167	1
SafetySynth	164	0
Simple BDD Solver	164	0
SafetySynth (Alt)	163	0
AbsSynthe (S3)	161	4
AbsSynthe (S2)	151	0
SDF	149	0
AbsSynthe (S1)	147	0
Demiurge D1real	129	6
TermiteSAT	97	4

Table 7.4: SYNTCOMP'16: sequential realisability track

Solver	Solved	Unique
AbsSynthe P1	181	1
TermiteSAT Hybrid	180	0
TermiteSAT Portfolio	179	0
Demiurge P3real	156	5
AbsSynthe P3	148	0
AbsSynthe P2	141	0

Table 7.5: SYNTCOMP'16: parallel realisability track

7.3.2 Synthesis Competition Results

I report the results of the sequential realisability track of the 2016 synthesis competition [SYNTCOMP, 2016] in Table 7.4 and the parallel realisability track in Table 7.5. The number of instances solved over time is shown in Figures 7.10 and 7.11 for the best configurations of each tool. The results show that although TERMITESAT does not perform as well in the sequential track, the combination of SAT based unbounded realisability with a traditional BDD solver performs very well in the parallel track. This supports the argument that different approaches are suited to different classes of specifications and that by combining solving techniques a greater coverage of problem instances is possible.

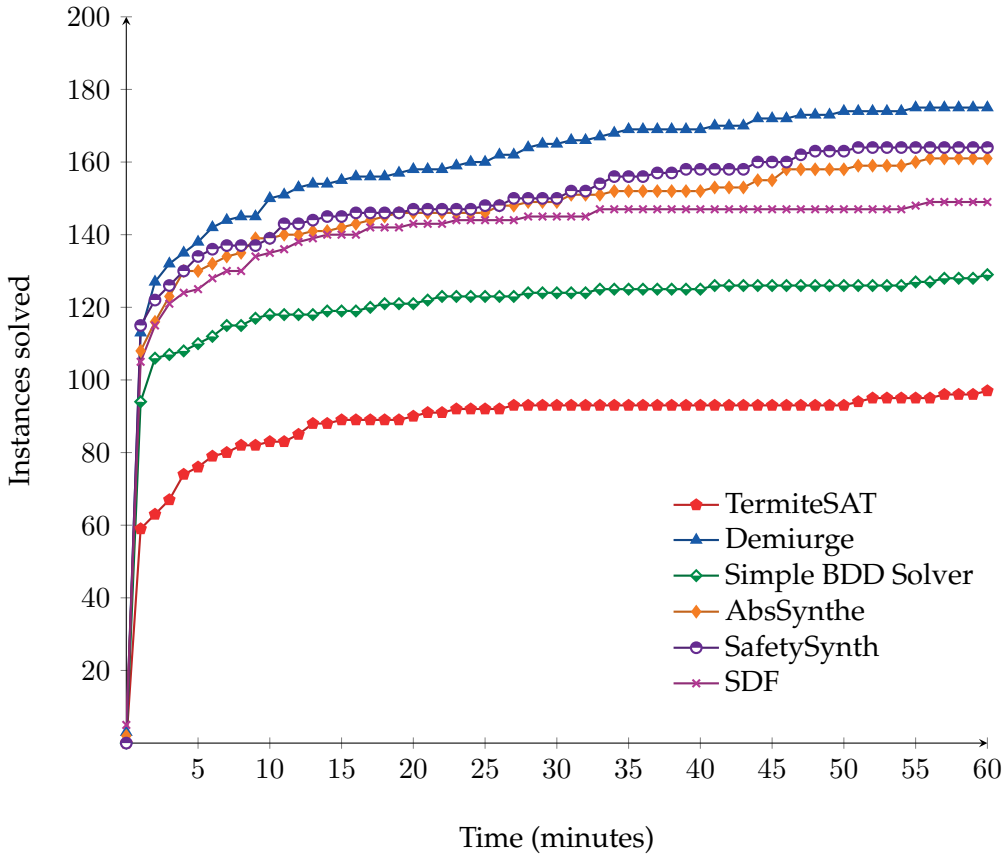


Figure 7.10: SYNTCOMP'16 sequential track: Instances solved over time

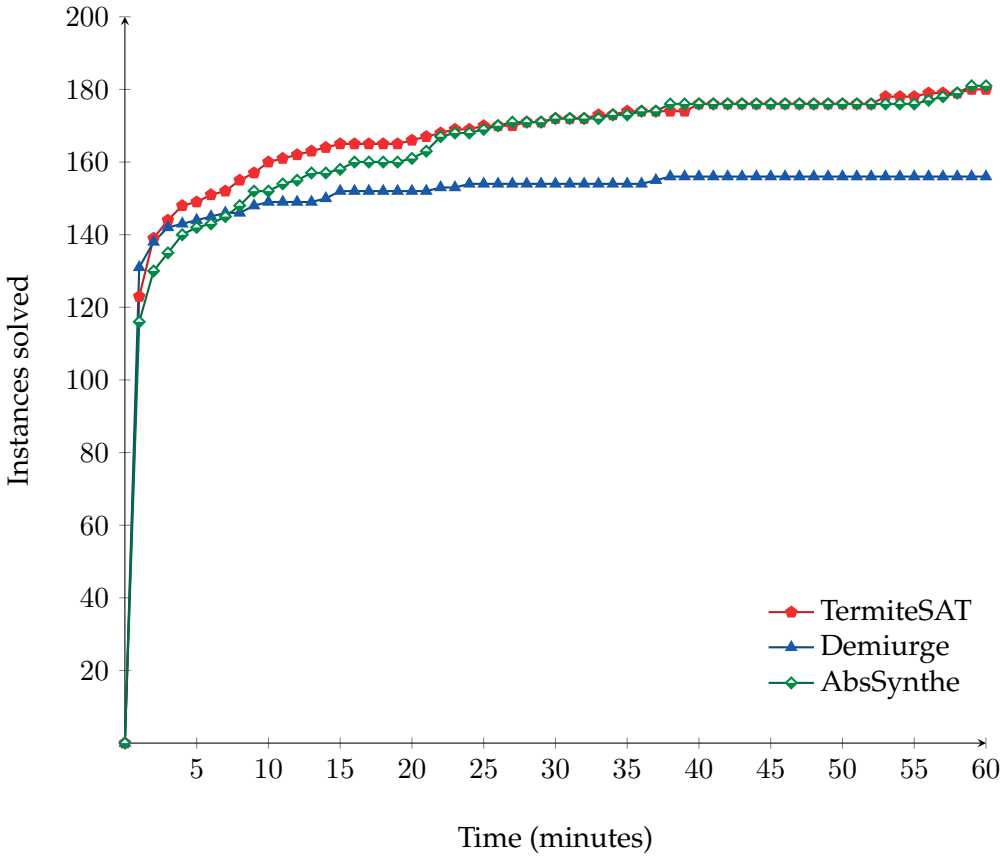


Figure 7.11: SYNTCOMP'16 parallel track: Instances solved over time

8 Conclusion

Controller synthesis may one day become the tool of choice for designers of reactive systems. As an approach to software correctness its advantages are clear but the future of synthesis is held back by computational complexity. In this thesis I presented a technique that takes one small step towards feasibility by focussing on a particular class of specifications.

The basis of this methodology is a counterexample guided bounded realisability algorithm. The algorithm constructs abstractions of the safety game and existentially searches for player strategies. The approach is similar to existing QBF methodologies but is able to perform better than those by taking advantage of knowledge of the structure of the problem. Bounded realisability is implemented in two tools: EVASOLVER and TERMITESAT. The latter of these is open source and available online. The approach was evaluated by benchmarking EVASOLVER against two QBF solvers: DEPQBF and RAREQS, as well as a BDD-based driver synthesis tool TERMITE. The results showed that bounded synthesis was able to outperform the BDD methods on classes of specifications for which the BDD representation of the winning region is exponential in the number of state variables.

I also presented an algorithm to extract the strategy from the certificate generated during bounded realisability. A certificate tree contains all of the actions required to prove that the game is realisable but does not contain enough information to construct a player strategy. I proposed an approach that uses Craig interpolation construct the set of states from which a player should choose a particular action in the certificate tree. Strategy extraction was implemented as part of EVASOLVER and benchmarking showed that it introduces a linear overhead.

Additionally I presented an approach that extends the algorithm to solve

unbounded games. The bounded realisability algorithm is able to show that there is no counterexamples traces of a certain length. In order to show that there is no counterexample of any length I proposed an approach that overapproximates the set of winning states for the controller. Overapproximation is achieved by carefully constructing interpolants in such a way that maintains two invariants that ensure that a fixed point in the computation is sufficient to prove realisability. Similar to the bounded realisability algorithm, this unbounded realisability approach can perform well in cases where a compact BDD representation of the winning region does not exist.

I implemented unbounded realisability in the open source TERMITESAT tool and submitted it to the synthesis competition. Although the submission was not competitive by itself it was able to solve several problem instances that no other solver could. I also implemented a hybrid approach within the tool that runs unbounded realisability in parallel with a BDD solver and shares some learned states. This hybrid approach performed very well in the parallel track of the competition. These results support the argument that unbounded synthesis is useful as part of a complete synthesis toolkit in order to solve problem instances that are infeasible for BDD based approaches.

In summary, I have presented a synthesis algorithm inspired by counterexample guided QBF and bounded model checking. I implemented the algorithm and demonstrated its usefulness in cases where traditional approaches are infeasible.

List of Figures

List of Tables

Bibliography

SYNTCOMP 2015 results. http://syntcomp.cs.uni-saarland.de/syntcomp2015/experiments/, a. Accessed: 2016-09-12.

SYNTCOMP 2016 results. http://syntcomp.cs.uni-saarland.de/syntcomp2016/experiments/, b. Accessed: 2016-09-12.

Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In Susanne Graf and Michael Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 411–425, Berlin, Germany, 2000. Springer Berlin Heidelberg.

Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In Panos Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems II*, chapter 1, pages 1–20. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-47519-4.

Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pages 399–404, Pasadena, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

Abdelwaheb Ayari and David A. Basin. QUBOS: Deciding quantified boolean logic using propositional satisfiability solvers. In Mark D. Aagaard and John W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 187–201, Portland, OR, USA, 2002. Springer Berlin Heidelberg.

Armin Biere. Resolve and expand. In Holger H. Hoos and David G. Mitchell, editors, *Revised Selected Papers of the 7th International Conference on Theory*

and Applications of Satisfiability Testing, volume 3542 of *Lecture Notes in Computer Science*, pages 59–70, Vancouver, BC, Canada, 2005. Springer Berlin Heidelberg.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207, London, UK, UK, 1999. Springer Berlin Heidelberg.

Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 101–115, Wrocław, Poland, 2011. Springer Berlin Heidelberg.

Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Könighofer, and Florian Lonsing. SAT-based methods for circuit synthesis. In Koen Claessen and Viktor Kuncak, editors, *Proceedings of the 14th International Conference on Formal Methods in Computer-Aided Design*, pages 31–34, Lausanne, Switzerland, 2014. FMCAD Inc.

Bart Bogaerts, Tomi Janhunen, and Shahab Tasharrofi. Solving QBF instances with nested SAT solvers. In *Beyond NP Workshop 2016 at the 30th International Conference on Artificial Intelligence*, Phoenix, Arizona, USA, 2016.

Aaron R. Bradley. SAT-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87, Austin, TX, USA, 2011. Springer Berlin Heidelberg.

Romain Brenguier, Guillermo A. Pérez, Jean-François Raskin, and Ocan Sankur. AbsSynthe: Abstract synthesis from succinct safety specifications. In Krishnendu Chatterjee, Rüdiger Ehlers, and Susmit Jha, editors, *Proceedings of the 3rd Workshop on Synthesis*, pages 100–116, Vienna, Austria, 2014.

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

J. Richard Büchi. On a decision method in restricted second order arithmetic. In Petr Hájek, Luis Valdés-Villanueva, and Dag Westerståhl, editors, *Proceedings of the 12th International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11, Oviedo, Spain, 1962. Stanford University Press.

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

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, USA, 1990. IEEE Computer Society.

Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference*, pages 262–267, Madison, WI, USA, 1998. AAAI Press / The MIT Press.

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 Diana Marculescu and Frank Liu, editors, *Proceedings of the International Conference on Computer-Aided Design*, pages 794–801, Austin, TX, USA, 2015. IEEE Computer Society.

Alonozo Church. Logic, arithmetic and automata. In V. Stenström, editor, *Proceedings of the International Congress of Mathematicians*, pages 23–35, Stockholm, Sweden, 1962. Almqvist & Wiksell.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Proceedings of the 12 International Conference on Computer Aided Verification*, pages 154–169, Chicago, IL, USA, 2000. Springer Berlin Heidelberg.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen,

editor, *Workshop on Logics of Programs*, pages 52–71, Yorktown Heights, NY, USA, 1981. Springer Berlin Heidelberg.

Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions Programming Languages Systems*, 8(2):244–263, 1986.

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

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

Mark Dowson. The Ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2), 1997.

Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

Niklas Eén, Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. SAT-based strategy extraction in reachability games. In Blai Bonet and Sven Koenig, editors, *Proceedings of the 29th Conference on Artificial Intelligence*, pages 3738–3745, Austin, TX, USA, 2015. AAAI Press.

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, *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 8312 of *Lecture Notes in Computer Science*, pages 291–308, Almaty, Kazakhstan, 2013. Springer Berlin Heidelberg.

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 Viktor Kuncak Andrey Rybalchenko, editor, *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 219–234, Philadelphia, PA, USA, 2012. Springer Berlin Heidelberg.

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 18th Conference on Artificial Intelligence*, pages 649–654, Edmonton, Alberta, Canada, 2002. AAAI Press / The MIT Press.

Alexandra Goultiaeva, Vicki Iverson, and Fahiem Bacchus. Beyond CNF: A circuit-based QBF solver. In Oliver Kullmann, editor, *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, volume 5584 of *Lecture Notes in Computer Science*, pages 412–426, Swansea, UK, 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 *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 546–553, Barcelona, Spain, 2011. AAAI Press.

Susanne Graf and Hassen Saidi. Construction of abstract state graphs of infinite systems with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, 1997. Springer Berlin Heidelberg.

Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. Sat-based image computation with application in reachability analysis. In Warren A. Hunt and Steven D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 391–408, Austin, TX, USA, 2000. Springer Berlin Heidelberg.

Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-guided control. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 886–902, Eindhoven, The Netherlands, 2003. Springer.

Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. The first reactive synthesis competition (SYNTCOMP 2014). *Computing Research Repository*, 2015.

Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. *Solving QBF with Counterexample Guided Refinement*, volume 7317 of *Lecture Notes in Computer Science*, pages 114–128. Springer Berlin Heidelberg, Trento, Italy, 2012.

Mikoláš Joao Marques-Silva Janota. On propositional QBF expansions and q-resolution. In Matti Järvisalo and Allen Van Gelder, editors, *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing*, volume 7962 of *Lecture Notes in Computer Science*, pages 67–82, Helsinki, Finland, 2013.

Mikoláš Janota and Joao Marques-Silva. Solving QBF by clause selection. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, pages 325–331, Buenos Aires, Argentina, 2015. AAAI Press.

William Klieber, Samir Sapra, Sicun Gao, and Edmund Clarke. A non-prenex, non-clausal QBF solver with game-state learning. In Ofer Strichman and Stefan Szeider, editors, *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, volume 6175 of *Lecture Notes in Computer Science*, pages 128–142. Springer, Springer Berlin Heidelberg, 2010.

Dexter Kozen. Results on the propositional μ-calculus. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 348–359, Aarhus, Denmark, 1982. Springer Berlin Heidelberg.

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

Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th Annual Design Automation Conference*, pages 263–268, Anaheim, CA, USA, 1997. ACM.

Alexander Legg. TermiteSAT. `http://www.github.com/alexlegg/TermiteSAT`, 2016.

Alexander Legg, Nina Narodytska, and Leonid Ryzhyk. A SAT-based counterexample guided method for unbounded synthesis. In Swarat Chaudhuri and Azadeh Farzan, editors, *Proceedings of the 28th International Conference on Computer Aided Verification*, volume 9780 of *Lecture Notes in Computer Science*, pages 364–382, Toronto, ON, Canada, 2016. Springer.

Florian Lonsing and Armin Biere. Integrating dependency schemes in search-based QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, volume 6175 of *Lecture Notes in Computer Science*, pages 158–171. Springer Berlin Heidelberg.

Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264, Copenhagen, Denmark, 2002.

Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13, Boulder, CO, USA, 2003. Springer Berlin Heidelberg.

Kenneth L. McMillan. Applications of craig interpolants in model checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12, Edinburgh, UK, 2005. Springer Berlin Heidelberg.

Andreas Morgenstern, Manuel Gesell, and Klaus Schneider. Solving games using incremental induction. In Einar Broch Johnsen and Luigia Petre, editors, *Proceedings of the 10th International Conference on Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, pages 177–191, Turku, Finland, 2013. Springer Berlin Heidelberg.

Nina Narodytska, Alexander Legg, Fahiem Bacchus, Leonid Ryzhyk, and Adam Walker. Solving games without controllable predecessor. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 533–540, Vienna, Austria, 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 Luca Benini, Giovanni De Micheli, Bashir M. Al-Hashimi, and Wolfgang Müller, editors, *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1596–1601, Nice, France, 2009. IEEE.

Florian Pigorsch and Christoph Scholl. An AIG-based QBF-solver using SAT for preprocessing. In *Proceedings of the 47th Annual Design Automation Conference*, pages 170–175, Anaheim, CA, USA, 2010. ACM.

Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive (1) designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.

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

Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 179–190, Austin, TX, USA, 1989. ACM.

Markus N. Rabe and Leander Tentrup. CAQE: A certifying QBF solver. In Roope Kaivola and Thomas Wahl, editors, *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, pages 136–143, Austin, TX, USA, 2015. IEEE.

Simone Fulvio Rollini, Leonardo Alt, Grigory Fedyukovich, Antti Eero Johannes Hyvärinen, and Natasha Sharygina. PeRIPLO: A framework for

producing effective interpolants in sat-based software verification. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 8312 of *Lecture Notes in Computer Science*, pages 683–693, Stellenbosch, South Africa, 2013. Springer.

Fabio Somenzi. CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`, 2001.

Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham Birtwistle, editors, *Proceedings of the 8th Higher Order Workshop on Logics for Concurrency - Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266, Banff, Alberta, Canada, 1996. Springer Berlin Heidelberg.

Adam Walker. Simple BDD Solver. `http://www.github.com/adamwalker/syntcomp`, 2014.

Adam Walker and Leonid Ryzhyk. Predicate abstraction for reactive synthesis. In Koen Claessen and Viktor Kuncak, editors, *Proceedings of the 14th International Conference on Formal Methods in Computer-Aided Design*, pages 219–226, Lausanne, Switzerland, 2014. FMCAD Inc.

Poul F. Williams, Armin Biere, Edmund M. Clarke, and Anubhav Gupta. *Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking*, volume 1855 of *Lecture Notes in Computer Science*, pages 124–138. Springer Berlin Heidelberg, Chicago, IL, USA, 2000.

Lintao Zhang and Sharad Malik. *Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation*, volume 2470 of *Lecture Notes in Computer Science*, pages 200–215. Springer Berlin Heidelberg, Ithaca, NY, USA, 2002.