Addressing State Explosion in Reactive Synthesis

Alexander Legg

Supervised by
Leonid Ryzhyk, Nina Narodytska, and Gernot Heiser

July 30, 2016

Contents

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 meaning that many specifications are infeasible to synthesise.

Synthesis is hard because reactive systems have many possible states that the synthesis procedure must consider. A standard way to mitigate this complexity is to symbolically represent sets of states with data structures such as binary decision diagrams (BDDs). The usual approach to synthesis is to construct the set of all states that are safe for the controller. In some specifications this set has no compact representation as a BDD and so it is slow or impossible to synthesise a controller in this way.

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 enables synthesis of specifications that were previously infeasible.

Acknowledgements

Acknowledge some people

Publications

List publications

1 Introduction

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

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

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

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

Unfortunately, automatic software synthesis involves nontrivial computa-

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

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

1.1 Synthesis

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

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

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

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

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

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

1.2 Approach

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

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

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

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

1.3 Contribution

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

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

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

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

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

1.4 Summary

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

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

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

2 Background

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

2.1 Temporal Logic

In this thesis we are concerned with reactive systems. Traditional programs can be verified by checking that the output is correct for each possible input. In a reactive system, i.e. a system that is in a continuous state of interaction with its environment, there is no termination and therefore no final output to verify. Instead the system is considered correct if it adheres to its specification indefinitely. A formalism of a reactive system must then consider the concept of time in order to specify its correctness property.

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

2.1.1 Kripke Structures

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

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

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

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

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

2.1.2 Linear Temporal Logic

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

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

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

- ϕ is a propositional formula referring to the current state,
- $X\phi$ - ϕ is true in the next state of the execution,
- $F\phi$ - Eventually (finally) ϕ will be true, and
- $G\phi$ - ϕ is always (globally) true.
- $\phi_1 U \phi_2$ - ϕ_1 holds until ϕ_2 holds.

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

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

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

2.1.3 Computation Tree Logic

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

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

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

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

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

2.2 Model Checking

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

The first such automatic model checker was proposed by Clarke et al. [Clarke et al., 1986] to verify temporal properties of finite state programs. The algorithm they proposed was a search based labelling of a finite state transition graph, representing the program, with subformulas of the temporal logic specification.

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

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

2.2.1 Büchi Automata

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

- The empty language , *or*
- A singleton language $\{a\}$ for $a \in \Sigma$, *or*
- For two regular languages A and B:

  - $A \cup B$ the union of those languages, *or*
  - $A \,.\, B$ the concatenation of those languages, *or*
  - $A*$ the Kleene operation on that language.

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

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma \to Q$ is a transition function mapping states and letters to next states,
- $q_0 \in Q$ is an initial state, and
- $F \subseteq Q$ is a set of accepting states. A accepts an input stream iff it visits F infinitely often.

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

2.3 Synthesis

Model checking is the art of deciding whether a program meets a specification. Synthesis is the related problem of *constructing* the program to meet a specification. In model checking the actions of the program are decided by the software being checked. A synthesis procedure must instead decide how the controller chooses actions and the ways the environment can react to each decision. In order to model this requirement the controller and environment should be considered to be in an adversarial relationship. Thus the synthesis problem is formulated as a two player game between the reactive program and its environment [Pnueli and Rosner, 1989].

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

- S is a finite set of states,

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

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

In addition to the game structure itself we define a game *objective* ψ given by an LTL formula. We say that a run is *winning* for the controller iff the run satisfies the objective. The game is zero-sum, therefore a run is winning for the environment in the dual case where the objective ψ is not true. For a controller to be correct it must ensure that all choices of the environment lead to runs that are winning for the controller.

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

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

2.3.1 Solving Games

Reactive synthesis for a game with an LTL objective [Pnueli and Rosner, 1989] may be solved via the construction of an equivalent non-deterministic Büchi automaton that is subsequently determinised to a deterministic Rabin automaton. Without delving into details, the Rabin automaton is interpreted as a

tree-automaton and checked for emptiness. This yields a double exponential time algorithm in the size of the specification.

The double exponential complexity causes a *state explosion*, which led to synthesis being considered infeasible for many years. However, synthesis has been applied in many real world scenarios by restricting the game objective to fragments of LTL. In this thesis we consider *safety games*, or games with objectives of the form $G\phi$ where ϕ is a propositional formula only. Informally, a controller in a safety game has the objective to stay within a safe region or avoid error states. Safety games can be solved by drawing on μ-calculus and solving a fixed point calculation [Asarin et al., 1995], which is a much less complex procedure than the above automata theoretic approach.

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

2.3.2 Fixed Point Computation

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

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

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

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

2.3.3 Fixed Point Game Solving

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

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

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

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

2.3.4 Symbolic Algorithms

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

Consider the game structure in the previous sections. Without loss of generality, we may replace the set of states S with a set of boolean variables. A single state is now a valuation to those variables and a set of states may be symbolically represented by a boolean function. In order to combat state explosion a compact representation of boolean functions is required. The standard choice is an ordered binary decision diagram (BDD) [Bryant, 1986].

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

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

2.3.5 Abstraction

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

An abstraction is a mapping of *concrete* states onto a new, smaller, set of *abstract* states. Abstractions may be created manually or automatically constructed. A common technique is to approximate an abstraction for a system and refine it during the verification process. Counterexample guided abstraction refinement (CEGAR) [Clarke et al., 2000] is a framework in which an approximate abstraction is refined via the analysis of counterexamples to the specification. An upper approximation is used for abstraction so that when the specification holds for the abstraction it also holds for concrete system. However, when a counterexample is found in the abstraction it may not be a concrete counterexample, in which case we call it a *spurious* counterexample. These counterexamples are used for refinement and the procedure begins anew with the refined abstraction.

2.4 Boolean Satisfiability

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

A CNF formula is a set of *clauses* of the form $(l_0 \vee l_1 \vee ... \vee l_n)$ where each *literal* l_i is a boolean variable or its negation. We call a clause with only one literal a *unit clause*. We call a variable *pure* if it appears in only one polarity in the formula. The DPLL algorithm propagates unit clauses and removes

clauses with pure literals as its first step. Next a value is assigned to a variable and the algorithm recurses to search for a solution with that valuation. If none can be found then a backtracking occurs and the other polarity of the variable is tried. In modern solvers, clause learning is used to share information between different branches of the search tree. The algorithm terminates when the current variable assignment satisfies the formula, or the search space is exhausted.

2.4.1 Bounded Model Checking

In the previous section BDDs were discussed as a symbolic representation that compresses boolean functions and efficiently quantifies the function. A CNF representation of a function is not canonical and so does not necessarily suffer from the same state explosion problems as BDDs. Biere et al. [Biere et al., 1999] introduced a model checking methodology that takes advantage of CNF as a symbolic representation and utilises a SAT solver to efficiently operate on it. We discussed BDDs in the context of solving games, however the context of this section is model checking of CTL* properties. It suffices to say that BDDs are applicable to this area as well and have similar advantages and disadvantages.

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

One difficulty of this approach is choosing a maximum bound that is both *sufficient* to verify the program correct and *feasible* to compute the result of the propositional formula. For a finite state automaton, the *diameter* is the minimal length such that every reachable state can by reached by a path of that length or less. The diameter is sufficient for model checking but not always feasible. In addition, computing the diameter itself is an inefficient quantified boolean formula. Despite this difficulty, bounded model checking is useful in many

practical cases. In particular the ability to quickly find short counterexamples gives an advantage in cases when a BDD based approach hits state explosion issues.

2.4.2 Interpolation

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

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

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

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

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 correctness condition that lays out a set of states of the system that controller must stay within. In this chapter we will refer to this property in the negation: the controller must avoid *error states*.

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

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

In realisability it is not enough to check for the existence of a trace that visits an error state. Such a trace only implies a counterexample that requires

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

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

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

4.1 Algorithm

This work draws inspiration from a QBF solving algorithm that treats the QBF problem as a game [Janota et al., 2012]. In that algorithm one player assumes the role of the universal quantifiers and the opponent takes on the existential quantifiers. In the game, the players take turns to chooses values for their variables from the outermost quantifier block in. Quantifiers may be removed from a formula by iteratively constructing and merging copies of the formula for each quantified variable. The copies of the formula represent the two possible values, true or false, of the quantified boolean variable. Universal

quantification can then be reduced to the conjunction of these copies, and existential quantifications corresponds to a disjunction. In practice, these expanded formulas are far too large to be solved so the authors introduce abstractions, or partially expanded formulas, to avoid expanding on variables unnecessarily. The abstractions are refined through a CEGAR process of searching for candidate solutions and analysing counterexamples. The full algorithm is described in detail in Chapter **??**.

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

It is possible to solve this QBF naïvely but we can do better by taking into account the inherent structure that arises in formulas constructed from realisability problems. The games we solve are deterministic so we know that state variables at a particular game unrolling are fully dependent on the action variables that come before. This enables a fine grained computational learning optimisation that would be difficult to reproduce in a general QBF algorithm.

The bounded realisability algorithm presented here takes inspiration from the CEGAR framework of the work of Janota et. al. and can be thought of as a domain specific version of the QBF algorithm.

4.2 Intuition

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

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

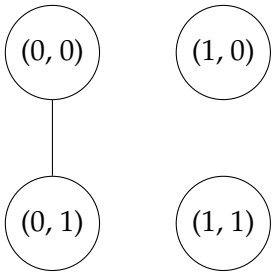


Figure 4.1: State automata representation of δ in Example 1.

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

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

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

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

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

Our bounded synthesis algorithm is set within a counterexample guided abstraction refinement framework. Abstraction serves a dual purpose in our approach as both a method for discovering player strategies 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 at its own strategy and the search is guided toward winning strategies.

We will now step through an execution of the algorithm using the example we have 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. A SAT solver can be used to efficiently do this search for us. Intuitively, an existential search is equivalent to the two players of the game cooperating. In most real world applications of synthesis the environment is in fact cooperating with the system so the cooperation heuristic turns out to be practical. Figure **??** shows a trace through the example game that reaches an error state.

The trace tells us that by playing the actions contained in the trace it is possible for the environment to reach an error state. From this we conjecture that the first action in the trace is a reasonable choice for the environment's winning strategy. So we construct a candidate strategy in which the environment plays `sdfsdf` 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 **??**). 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.

Our bounded synthesis algorithm constructs abstractions of the game that restrict actions available to one of the players. Specifically, we consider abstractions represented as trees of actions, referred to as *abstract game trees* (AGTs). Figure 5.2c shows an example abstract game tree restricting the environment (abstract game trees restricting the controller are similar). In the abstract game, the controller can freely choose actions whilst the environment is required to pick actions from the tree. After reaching a leaf, the environment continues playing unrestricted. The tree in Figure 5.2c restricts the first environment action to `request=1`. At the leaf of the tree the game continues unrestricted.

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

(a) Variables

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

(b) Initial State

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

(c) Transition Relation

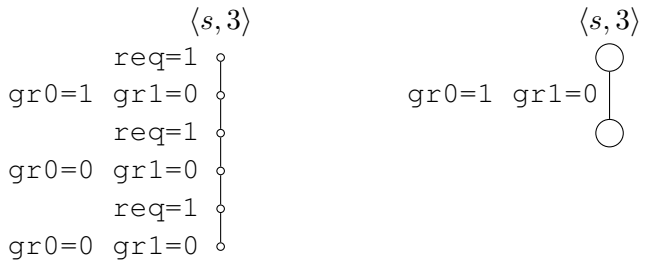
Figure 4.2: Example

4.3 Case Study: Arbiter

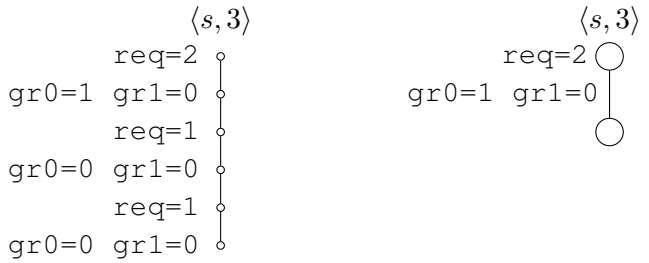
We consider a simple arbiter system in which the environment makes a request for a number of resources (1 or 2), and the controller may grant access to up to two resources. The total number of requests grows each round by the number of environment requests and shrinks by the number of resources granted by the controller in the previous round. The controller must ensure that the number of unhandled requests does not accumulate to more than 2. Figure 4.2 shows the variables (4.2a), the initial state of the system (4.2b), and the formulas for computing next-state variable assignments (4.2c) for this example. We use primed identifiers to denote next-state variables and curly braces to define the domain of a variable.

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

Our bounded synthesis algorithm constructs abstractions of the game that restrict actions available to one of the players. Specifically, we consider abstractions represented as trees of actions, referred to as *abstract game trees* (AGTs). Figure 5.2c shows an example abstract game tree restricting the environment



(a) Controller winning trace (b) AGT



(c) Environment winning trace (d) Partial Strategy

Figure 4.3: Abstract game trees.

(abstract game trees restricting the controller are similar). In the abstract game, the controller can freely choose actions whilst the environment is required to pick actions from the tree. After reaching a leaf, the environment continues playing unrestricted. The tree in Figure 5.2c restricts the first environment action to `request=1`. At the leaf of the tree the game continues unrestricted.

The root of the tree is annotated by the initial state s of the abstract game and the bound k on the number of rounds. We denote NODES(T) the set of all nodes of a tree T, LEAVES(T) the subset of leaf nodes. For edge e, ACTION(e) is the action that labels the edge, and for node n, HEIGHT(k, n) is the distance from n to the last round of a game bounded to k rounds. HEIGHT(k, T) is the height of the root node of the tree. For node n of the tree, SUCC(n) is the set of pairs $\langle e, n' \rangle$ where n' is a child node of n and e is the edge connecting n and n'.

Given an environment (controller) abstract game tree T a *partial strategy* $Strat : \text{NODES}(T) \to \mathcal{C}$ ($Strat : \text{NODES}(T) \to \mathcal{U}$) labels each node of the tree with the controller's (environment's) action to be played in that node. Given a partial strategy $Strat$, we can map each leaf l of the abstract game tree to $\langle s', i' \rangle = \text{OUTCOME}(\langle s, i \rangle, Strat, l)$ 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.

Example: Intuition behind the algorithm. *We present the intuition behind our bounded synthesis method by applying its* simplified version *to the running example. We begin by finding a trace of length k (here we consider $k = 3$) that is winning for the controller, i.e., that starts from the initial state and avoids the error set for three game rounds (see Figure 4.3a). We use a SAT solver to find such a trace, precisely as one would do in bounded model checking. Given this trace we make an initial conjecture that any trace starting with action* `gr0=1 gr1=0` *is winning for the controller. This conjecture is captured in the abstract game tree shown in Figure 5.2c. We validate this conjecture by searching for a counterexample trace that reaches an error state with the first controller action fixed to* `gr0=1 gr1=0`*. Such a trace, that refutes the conjecture, is shown in Figure 4.3c. In this trace, the environment wins by playing* `req=2` *in the first round. This move represents the environment's partial strategy against the abstract game tree in Figure 5.2c. This partial strategy is shown in Figure 4.3d.*

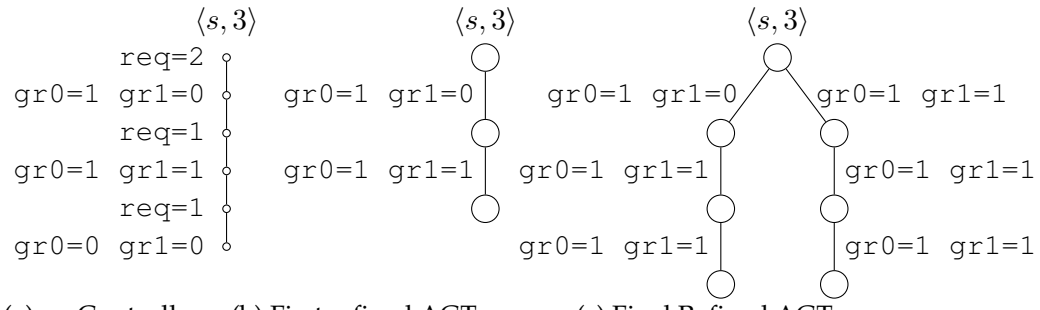
Next we strengthen the abstract game tree taking this partial strategy into account. To this end we again use a SAT solver to find a trace where the contoller wins while the environment plays according to the partial strategy. In the resulting trace (Figure 4.4a), the controller plays `gr0=1 gr1=1` *in the second round. We refine the abstract game tree using this move as shown in Figure 4.4b. The environment's partial strategy was to make two requests in the first round, to which the controller responds by now granting an additional two resources in the second round.*

When the controller cannot refine the tree by extending existing branches, it backtracks and creates new branches. Eventually, we obtain the abstract game tree shown in Figure 4.4c for which there does not exist a winning partial strategy on behalf of the environment. We conclude that the bounded game is winning for the controller.

The full bounded synthesis algorithm is more complicated: upon finding a candidate partial strategy on behalf of player p against abstract game tree T, it first checks whether the strategy is winning against T. By only considering such strong candidates, we reduce the number of refinements needed to solve the game. To this end, the algorithm checks whether each outcome of the candidate strategy is a winning state for $\text{OPPONENT}(p)$ by recursively invok-

Algorithm 1 Bounded synthesis

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



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

Figure 4.4: Refined abstract game trees.

ing the synthesis algorithm on behalf of the opponent. Thus, our bounded synthesis algorithm can be seen as running two competing solvers, for the controller and for the environment.

The full procedure is illustrated in Algorithm 1. The algorithm takes a concrete game G with maximum bound κ as an implicit argument. In addition, it takes a player p (controller or environment), state s, bound k and an abstract game tree T and returns a winning partial strategy for p, if one exists. The initial invocation of the algorithm takes the initial state I, bound κ and an empty abstract game tree $\emptyset$. Initially the solver is playing on behalf of the environment since that player takes the first move in every game round. The empty game tree does not constrain opponent moves, hence solving such an abstraction is equivalent to solving the original concrete game.

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

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

If the dual solver can find no spoiling strategy at any of the leaves, then the candidate partial strategy is a winning one. Otherwise, VERIFY returns the move used by the opponent to defeat a leaf of the partial strategy, which is

appended to the corresponding node in T in order to refine it in line (9).

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

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

The tree is first extended to the maximum bound with edges that are labeled with arbitrary opponent actions (Algorithm 1, line 14). For each node in the tree, new SAT variables are introduced corresponding to the state (X_T) and action (U_T or C_T) variables of that node. Additional variables for the opponent actions in the edges of T are introduced (U_e or C_e) and set to 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.

4.4 Optimisations

In this section we present several optimisations to the algorithm.

4.4.1 Bad State Learning

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

Algorithm 2 Tree formulas for Controller and Environment respectively

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

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

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

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

  end if
end function

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

Additional states can be learned by expanding a single state into a set of

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

Algorithm 3 Modified Tree Formulas with Bad State Avoidance

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

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

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

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

  end if
end function

4.4.2 Strategy Shortening

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

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

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

Algorithm 4 Strategy Shortening

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

4.4.3 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.

4.5 Correctness

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

4.6 Discussion

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

4.7 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 **??** I will present results and an evaluation of the contribution of this.

- 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.

5.1 Algorithm

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

We use the certificate tree computed by the game solver as a starting point for strategy generation. We know that the controller can win the game in κ rounds by picking actions from the tree; however we do not yet know exactly which actions should be picked in every state.

Example 1. *Figure 5.1 introduces the running example for this chapter. It shows a state machine for a game $(S, \mathcal{U}, \mathcal{C}, \delta, s_0)$ where S consists of two one bit variables $\mathtt{p}$ and $\mathtt{e}$, $\mathcal{U}$ and $\mathcal{C}$ each consist of a single one bit variable $\mathtt{u}$ and $\mathtt{c}$ respectively. The*

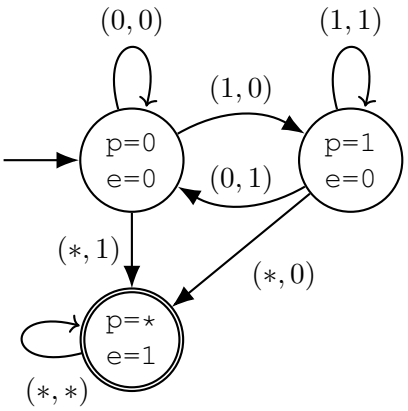


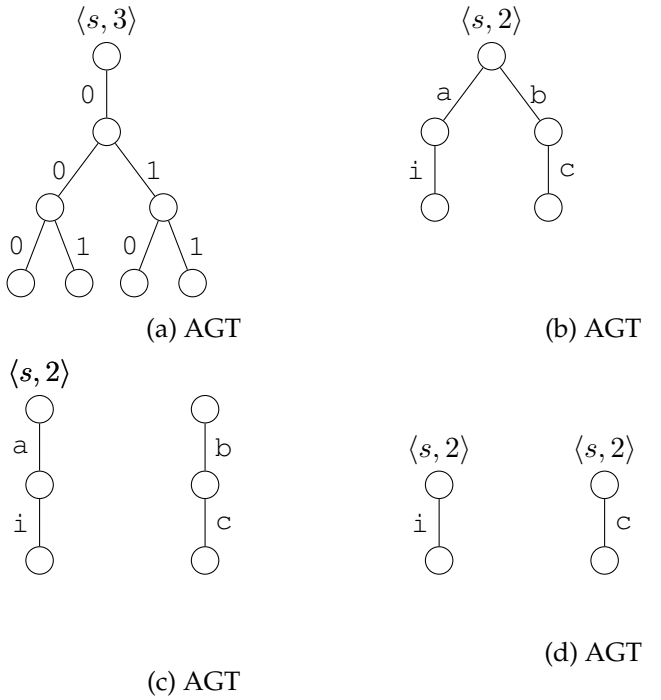
Figure 5.1: Transition relation of the running example

diagram shows δ as a deterministic finite state automaton with edges labelled by a tuple (u, c). We use $\star$ as a wildcard value to simplify the presentation. The game is simple, p is set to the previous value of u and e is set to 1 when $c \neq p$. The initial state, s_0, is $p = 0 \wedge e = 0$ and the error set is $e = 1$. In other words, the controller must simply match the previous action of the environment to avoid the error set.

The strategy generation algorithm takes an initial set s and its certificate tree T, computed by the game solver, and generates a winning controller strategy starting from s. To this end, it first partitions s into subsets s_i, one for each outgoing branch of T (Figure 5.3), such that the controller can win from s_i by picking action a_i along the ith branch of the tree. This partitioning defines the local winning strategy in the root node of T. Next, for each partition s_i, the algorithm computes the set of a_i-successor states of s_i, obtaining the set s'_i of states winning in the child subtree T'_i of T_i (Figure **??**). The algorithm is then invoked recursively for each subtree T'_i.

Example 2. *Figure 5.2 illustrates operation of the algorithm on the winning abstract game tree returned by the game solver for our running example (Figure 5.1). The algorithm starts at the root of the tree and the initial set $I = \{x_0\}$. The game tree only defines one winning action in the root node, hence this action is winning in all states of I and no partitioning is required. We compute the successor set reachable by playing action i in I: $I' = succ(I, i) = \{s_1, s_2\}$.*

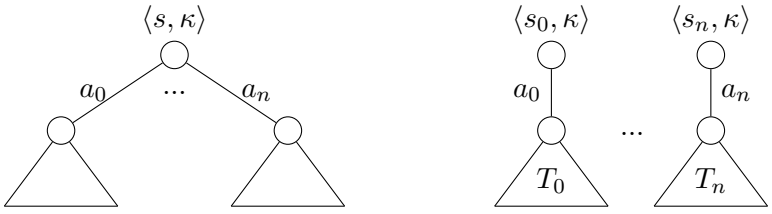
Next, we descend down the i branch of the tree and consider subtree T' and its initial set I' (Figure 5.2(b)). We partition I' into subsets $I'_1 = \{s_1\}$ and $I'_2 = \{s_2\}$ that are winning for the left and right subtrees of T' respectively, i.e., the controller



(a) AGT (b) AGT

(c) AGT (d) AGT

Figure 5.2: Operation of the strategy extraction algorithm on the example



(a) Before (b) After

Figure 5.3: Partitioning

must play action a in state s_1 and b in s_2. Consider the resulting subtrees T_1' and T_2' with initial sets I_1' and I_2' (Figure 5.2(c)). We have $I_1'' = succ(I_1', a) = \{s_4\}$, $I_2'' = succ(I_2', b) = \{s_3\}$. Finally, we obtain two subtrees T_1'' and T_2'' with initial sets I_1'' and I_2'' (Figure 5.2(d)). Both subtrees have one branch; hence corresponding actions i and c are winning in I_1'' and I_2'' respectively.

Putting together fragments of the winning strategy computed above, we obtain the following strategy for this example: $\pi(s_0) = i$, $\pi(s_1) = a$, $\pi(s_2) = b$, $\pi(s_3) = c$, $\pi(s_4) = i$.

The above algorithm involves two potentially costly operations: winning set partitioning and successor set computation. If implemented naïvely, these operations can lead to unacceptable performance. The key insight behind our solution is that both operations can be efficiently approximated from the proof of unsatisfiability of the formula $E_T(I)$, with the help of interpolation, as described below. The resulting approximations are sound, i.e., preserve the correctness of the resulting strategy.

Algorithm 5 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, I) into j pairs (T_i, I_i), as shown in Figure **??**. Each tree T_i is a copy of a single branch of T. The partitioning is constructed in such a way that the action a_i that labels the root edge of T_i is a winning controller action in I_i.

Next we consider each pair (T_i, I_i) (lines 10-14). We descend down the tree and compute the controller strategy in the child subtree T_i' of T_i (right-hand side of Figure **??**). To do so, we first compute the set of a_i-successors of I_i: More precisely, we compute an overapproximation $I_i' \supseteq succ(I_i, a_i)$, such that T_i' is a certificate tree for I_i'. Such an overapproximation is returned by the NEXT function in line 11. We can now recursively invoke the strategy generation function to compute a winning strategy for the pair (T_i', I_i') (line 12).

The algorithm returns the set of tuples (W, a, k). Each tuple represents a fragment of the strategy in some tree node, where W is the winning set in this node, a is the controller action to play in this set, and k is the distance from the node to the bottom of the tree (i.e., distance to the goal).

Algorithm 5 Computing a winning strategy

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

function GENLOCALSTRATS(T, I)
  $v \leftarrow root(T)$
  $[(e_1, a_1, v_1), \ldots, (e_j, a_j, v_j)] \leftarrow edges(v)$
  $[(T_1, I_1), \ldots, (T_j, I_j)] \leftarrow$ PARTITION$(T, I \wedge \neg O)$
  $Strat \leftarrow \{(I_i, a_i, height(T)) \mid i \in [1, j]\}$
  for $i = 1$ to j **do**
    $(T_i', I_i') \leftarrow$ NEXT(T_i, I_i)
    $Strat_i \leftarrow$ GENLOCALSTRATS(T_i', I_i')
    $Strat \leftarrow Strat \cup Strat_i$
  end for
  return $Strat$
end function

Algorithm 6 Partitioning winning states

function PARTITION(T, I)
  $v \leftarrow root(T)$
  $\hat{I} \leftarrow I, \hat{T} \leftarrow T$
  for $i = 1$ to j **do**
    $(T_i, \tilde{T}) \leftarrow split(\hat{T})$
    $A \leftarrow E_{\tilde{T}}(\hat{I})$
    $B \leftarrow E_{T_i}(\top)$
    $\mathcal{I}(s_v) \leftarrow Interpolant(A, B)$
    $I_i \leftarrow \mathcal{I}(s) \wedge \hat{I}$
    $\hat{I} \leftarrow \hat{I} \wedge \neg\mathcal{I}(s),\ \hat{T} \leftarrow \tilde{T}$
  end for
  return $[(T_1, I_1), \ldots, (T_j, I_j)]$
end function

5.1.1 Partition

The PARTITION function (Algorithm 6) computes a local strategy in the root of an abstract game tree. It takes a pair (T, I), such that T is a certificate tree for set I and partitions I into subsets I_i such that the controller can win by choosing action a_i in I_i.

At every iteration, the algorithm splits the tree into the leftmost branch

T_i and the remaining tree (Figure **??**). It then computes the set I_i where the controller wins by following the branch T_i and removes I_i from the initial set I. At the next iteration it considers the leftover tree $\tilde{T}_i$ and the shrunk initial set $\hat{I}$.

The algorithm maintains the invariant that $\hat{T}$ is a certificate tree for $\hat{I}$ and hence $E_{\hat{T}}(\hat{I})$ is unsatisfiable. We decompose this formula into two conjuncts $E_{\hat{T}}(\hat{I}) = A \wedge B$ such that A and B only share state variables s_v in the root node v of T and that the interpolant $\mathcal{I}$ of A and B consists of states where the controller can win by following the T_i subtree. Hence $\mathcal{I} \wedge \hat{I}$ gives us the desired set I_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 $\hat{I}$ 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 their conjunction is unsatisfiable.

The interpolant $\mathcal{I}$ of A and B implies $\neg B$, i.e., for any state in $\mathcal{I}$, a_i is a winning move. $\mathcal{I}$ is also implied by A, i.e., it contains all states in I where the controller cannot win by picking moves from $\tilde{T}$ as a subset. Equivalently, for any state in $I_i \wedge \neg\mathcal{I}$, the controller *can* win by following $\tilde{T}$, i.e., $\tilde{T}$ is a certificate tree for $I_i \wedge \neg\mathcal{I}$, and we can apply the decomposition again to $\tilde{T}$ and $I_i \wedge \neg\mathcal{I}$ at the next iteration.

We prove useful properties of the PARTITION function. First, we show that A and B indeed form a conjunctive decomposition of $E_{\hat{T}}(\hat{I})$.

Proposition 1. $A \wedge B = E_{\hat{T}}(\hat{I})$.

Proof. $A \wedge B = E_{\tilde{T}}(\hat{I}) \wedge E_{T_i}(\top) = \hat{I} \wedge \psi_{root(\tilde{T})} \wedge \top \wedge \psi_{root(T_i)} = \hat{I} \wedge (\psi_{root(\tilde{T})} \wedge \psi_{root(T_i)}) = \hat{I} \wedge \psi_{\hat{T}} = E_{\hat{T}}(\hat{I})$. □ □

Proposition 2. *The following invariant is maintained throughout the execution of* PARTITION*: $\hat{T}$ is a certificate tree for $\hat{I}$.*

Proof. We prove by induction. The invariant holds for the initial assignments of $\hat{T}$ and $\hat{I}$. By Proposition 1 and induction hypothesis, $A \wedge B = E_{\hat{T}}(\hat{I})$ is unsatisfiable. Hence the interpolation operation in line 8 is well defined. By the properties of interpolants, $(A \implies \mathcal{I}(s_v))$, hence $(\neg\mathcal{I}(s_v) \implies \neg A)$ or equivalently $(\neg\mathcal{I}(s_v) \implies \neg E_{\tilde{T}}(\hat{I}))$.

After $\hat{T}$ and $\hat{I}$ are updated in line 10, their new values $\hat{T}'$ and $\hat{I}'$ satisfy the following equalities: $E_{\hat{T}'}(\hat{I}') = E_{\tilde{T}}(\hat{I} \wedge \neg\mathcal{I}(s)) = \neg\mathcal{I}(s_v) \wedge E_{\tilde{T}}(\hat{I}) \implies \neg E_{\tilde{T}}(\hat{I}) \wedge E_{\tilde{T}}(\hat{I}) = \bot$ and hence the invariant is maintained. □ □

Proposition 3. *Let T be a certificate tree for I and let $I \wedge O = \bot$. Then $[(T_1, I_1), \ldots, (T_j, I_j)] =$ PARTITION(T, I) is a local winning strategy in the root of T, i.e., the following properties hold:*

1. *Sets $I_1, \ldots, I_j$ comprise a partitioning of I: $I = \bigvee I_i$ and $\forall i, k.(i \neq k) \implies I_i \wedge I_k = \bot$*

2. *T_i is a certificate tree for I_i, for all $i \in [1, j]$*

Proof. At every iteration of the algorithm, we partition $\hat{I}$ into $I_i = \mathcal{I} \wedge \hat{I}$ and $\hat{I} \wedge \neg\mathcal{I}$; hence different sets I_i do not overlap by construction.

At the final iteration of the algorithm, the tree $\tilde{T}$ consists of a single root node without outgoing branches. Hence, $A = E_{\tilde{T}}(\hat{I}) = \hat{I}(s_v) \wedge \neg O(s_v) = \hat{I}(s_v)$. Since $(A \implies \mathcal{I}(s_v))$, we get $(\hat{I} \implies \mathcal{I}(s))$ and therefore $\mathcal{I}(s) \wedge \hat{I} = \hat{I}$, i.e., all states in $\hat{I}$ are included in the final set I_j and hence the partitioning completely covers set I: $I = \bigvee I_i$.

We prove the second statement of the proposition. The set I_i is computed as $\mathcal{I}(s) \wedge \hat{I}$ (line 9) at the ith iteration of the algorithm. Thus, $E_{T_i}(I_i) = E_{T_i}(\mathcal{I}(s) \wedge \hat{I}) = \mathcal{I}(s) \wedge \hat{I} \wedge E_{T_i}(\top)$. By the properties of interpolants, $\mathcal{I}(s) \wedge B = \mathcal{I}(s) \wedge E_{T_i}(\top) = \bot$. Hence $E_{T_i}(I_i) = \bot$, i.e., T_i is a certificate tree for I_i. □ □

5.1.2 Next

The NEXT function (Algorithm 7) takes a set I and its certificate tree T, such that there is exactly one outgoing edge, labelled a, from the root node of T. T has a sole child subtree T' with root node v'. The function computes an overapproximation I' of the a-successor of I, such that I' is winning for the controller and T' is a certificate tree for I'.

Once again, we decompose the unsatisfiable formula $E_T(I)$ into two conjuncts A and B. A encodes one round of the game from the set I, where the controller plays action a. $B = E_{T'}(\top)$ is a partial $\forall$-expansion of the game induced by T'. A and B only share state variables $s_{v'}$ and their interpolant gives the approximation we are looking for.

Algorithm 7 Successor set

```
1: function NEXT(T, I)
2:     T' ← subtree(T, 1)
3:     v ← root(T)
4:     [(e, a, v')] ← edges(v)
5:     A ← I(s_v) ∧ Δ(s_v, c_e, u_e, s_v') ∧ c_e = a
6:     B ← E_T'(⊤)
7:     I(s_v') ← Interpolant(A, B)
8:     return (T', I(s))
9: end function
```

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

1. *$\mathcal{I}$ is an overapproximation of the a-successor of I, i.e., $\mathcal{I} \supseteq succ(I, a)$*

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

Proof. We rewrite formula (**??**) in the symbolic form: $succ(I, a) = \exists s_v, u_e . I(s_v) \wedge \Delta(s_v, c_e, u_e, s_{v'}) \wedge c_e = a$. The matrix of this formula is exactly formula A. Hence $succ(I, a) = \exists s_v, u_e . A$. Since $(A \implies \mathcal{I}(s_{v'}))$, $succ(I, a) \implies \exists s, u . \mathcal{I}(s_{v'})$. Since $\mathcal{I}$ is defined over state variables only, the quantifiers can be removed: $succ(I, a) \implies \mathcal{I}(s_{v'})$ or, in the relational form, $\mathcal{I} \supseteq succ(I, a)$. We prove the second property: $E_{T'}(\mathcal{I}) = \mathcal{I}(s_{v'}) \wedge E_{T'}(\top) = \mathcal{I}(s_{v'}) \wedge B = \bot$. □ □

5.1.3 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 8) achieves this by keeping track of all states W already added to the strategy. For every new tuple (I, a, k), it restricts the set I to $\neg W$, which guarantees that no state can be added to the strategy twice. Furthermore, by considering tuples with smaller values of k first, we resolve the nondeterminism in a way that guarantees progress towards the goal at every round of the game.

Algorithm 8 Compiling the winning strategy

```
1: function CompileStrat(Strat)
2:     π ← ⊥,  W ← ⊥
3:     for (I, a, k) ∈ Strat do                ▷ Sorted by ascending k
4:         π ← π ∨ (I ∧ ¬W ∧ (c = a))
5:         W ← W ∨ I
6:     end for
7:     return π
8: end function
```

Let $rank(s), s \in S$, be the smallest k such that there exists $(I, a, k) \in Strat$, $s \in I$, or ∞ if there is no such k.

Proposition 5. *Let $\pi = \text{CompileStrat}(Strat)$. For any pair $(s, a) \in \pi$, there exists $(I, a, k) \in Strat$ such that $s \in I$ and $k = rank(s)$.*

Theorem 1 (Correctness of the algorithm)**.** *Let abstract game tree T of height n be a certificate tree for the set I, let π be a partial function returned by the strategy generation algorithm, $\pi = \text{GenStrategy}(T, I)$, and let π' be an arbitrary extension of π to a complete function. Then π' is a winning controller strategy of length n from I.*

Proof. Every state-action pair (s, a) in π is generated by the Partition function and, according to Proposition 3, a is a winning controller move in s. By Proposition 4, all possible a-successors of s are either goal states or are covered by π. Hence, by following the strategy π from I, the controller is guaranteed to stay within the winning region of the game until reaching the goal.

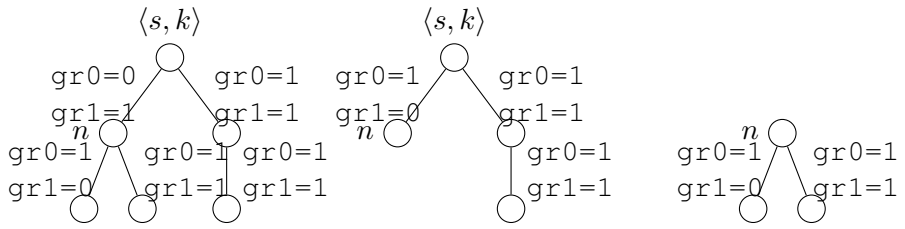
Next, we show that ranks of states visited by following the strategy π decrease monotonically and hence the strategy reaches the goal in at most n steps. According to Proposition 5, for every pair $(s, a) \in \pi$, there exists $(I, a, k) \in Strat$, such that $k = rank(s)$. Therefore, for any $s' \in succ(s, a)$, such that $s' \notin O$, there exists $(I', a', k-1) \in Strat$, $s' \in I'$; hence $rank(s') \leq k - 1 < k = rank(s)$. □ □

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 ($|\mathcal{X}|$) so that all states may be explored by the algorithm. A more nuanced approach is to use the diameter of the game [**?**], which is the smallest number d such that for any state x there is a path of length $\leq d$ to all other reachable states. Computing the diameter, however, is also expensive and solving a game bounded to the size of the diameter may be infeasible.

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



(a) A losing AGT T (b) Tree slice T_1 (c) Tree slice T_2

Figure 6.1: Splitting of an abstract game tree by the learning procedure.

6.0.1 Learning States with Interpolants

We extend the bounded synthesis algorithm to learn states losing for one of the players from failed attempts to find candidate strategies. The learning procedure kicks in whenever FINDCANDIDATE cannot find a candidate strategy for an abstract game tree. We can learn additional losing states from the tree via interpolation. This is achieved in lines 18–20 in Algorithm 1, enabled in the unbounded version of the algorithm, which invoke LEARN or $\overline{\text{LEARN}}$ to learn controller or environment losing states respectively (Algorithm 9).

Example: Why we use interpolants. *Consider node n in Figure 6.1a. 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.*

One option is using a BDD solver, working backwards from the error set, to find all losing states. One iteration of this operation on our example would give the set: $\texttt{nrequests} = 3 \vee (\texttt{nrequests} = 2 \wedge (\texttt{resource0} = 0 \vee \texttt{resource1} = 0)) \vee (\texttt{nrequests} = 1 \wedge (\texttt{resource0} = 0 \wedge \texttt{resource1} = 0))$*. In the general case there is no compact representation of the losing set, so we try to avoid computing it by employing interpolation instead. The benefit of interpolation is that it allows approximating the losing states efficiently by obtaining an interpolant from a SAT solver.*

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. An interpolant can be constructed efficiently from a resolution proof of the unsatisfiability of $F_1 \wedge F_2$ [**?**].

Algorithm 9 Learning algorithms

Require: $s(X_T) \wedge \text{TREEFORMULA}(k, T) \equiv \bot$
Require: *Must-invariant* holds
Ensure: *Must-invariant* holds
Ensure: $s(X_T) \wedge B^M \not\equiv \bot$ ▷ s will be added to B^M
1: **function** LEARN(s, T)
2: **if** SUCC(T) $= \emptyset$ **then return**
3: $n \leftarrow$ non-leaf node with min height
4: $\langle T_1, T_2 \rangle \leftarrow$ GTSPLIT(T, n)
5: $\mathcal{I} \leftarrow$ INTERPOLATE($s(X_T) \wedge$ TREEFORMULA(k, T_1), TREEFORMULA(k, T_2))
6: $B^M \leftarrow B^M \vee \mathcal{I}$
7: LEARN(s, T_1)
8: **end function**
Require: $s(X_T) \wedge \overline{\text{TREEFORMULA}}(k, T) \equiv \bot$
Require: *May-invariant* holds
Ensure: *May-invariant* holds
Ensure: $s(X_T) \wedge B^m[\text{HEIGHT}(k, T)] \equiv \bot$ ▷ s will be removed from B^m
9: **function** $\overline{\text{LEARN}}(s, T)$
10: **if** SUCC(T) $= \emptyset$ **then return**
11: $n \leftarrow$ non-leaf node with min height
12: $\langle T_1, T_2 \rangle \leftarrow$ GTSPLIT(T, n)
13: $\mathcal{I} \leftarrow$ INTERPOLATE($s(X_T) \wedge \overline{\text{TREEFORMULA}}(k, T_1)$, $\overline{\text{TREEFORMULA}}(k, T_2)$)
14: **for** $i = 1$ to HEIGHT(k, n) **do**
15: $B^m[i] \leftarrow B^m[i] \setminus \mathcal{I}$
16: **end for**
17: $\overline{\text{LEARN}}(s, T_1)$
18: **end function**

We choose a non-leaf node n of T with maximal depth, i.e., a node whose children are leafs (Algorithm 9, line 3). We then split the tree at n such that both slices T_1 and T_2 contain a copy of n (line 4). Figure 6.1b shows T_1, which contains all of T except n's children, and T_2 (Figure 6.1c), which contains only n and its children. There is no candidate strategy for T so $s \wedge \overline{\text{TREEFORMULA}}(k, T)$ is unsatisfiable. By construction, $\overline{\text{TREEFORMULA}}(k, T) \equiv \overline{\text{TREEFORMULA}}(k, T_1) \wedge \overline{\text{TREEFORMULA}}(k, T_2)$ and hence $s \wedge \overline{\text{TREEFORMULA}}(k, T_1) \wedge \overline{\text{TREEFORMULA}}(k, T_2)$ is also unsatisfiable.

We construct an interpolant with $F_1 = s(X_T) \wedge \text{TREEFORMULA}(k, T_1)$ 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.

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

We have discovered a set $\mathcal{I}$ of states losing for the environment. Environment-losing states are only losing for a particular bound: given that there does not exist an environment strategy that forces the game into an error state in k rounds or less; there may still exist a longer environment-winning strategy. We therefore record learned environment-losing states along with associated bounds. To this end, we maintain a conceptually infinite array of sets $B^m[k]$ that are may-losing for the controller, indexed by bound k. $B^m[k]$ are initialised to E for all k. Whenever an environment-losing set $\mathcal{I}$ is discovered for a node n with bound $\text{HEIGHT}(k, n)$ in line 13 of Algorithm 9, this set is subtracted from $B^m[i]$, for all i less than or equal to the bound (lines 14–16).

The $\overline{\text{TREEFORMULA}}$ function is modified for the unbounded solver (Algorithm 10) to constrain the environment to the appropriate B^m. This enables further interpolants to be constructed by the learning procedure recursively splitting more nodes from T_1 (Algorithm 9, line 7) since the states that are losing to T_2 are no longer contained in B^m.

Learning of states losing from the controller is similar (LEARN in Algorithm 9). The main difference is that environment-losing states are losing for all bounds. Therefore we record these states in a single set B^M of must-losing states (Algorithm 9, line 6). This set is initialised to the error set E and grows as new losing states are discovered. The modified $\overline{\text{TREEFORMULA}}$ function (Algorithm 10) blocks must-losing states, which also allows for recursive learning over the entire tree.

6.0.2 Main synthesis loop

Figure 11 shows the main loop of the unbounded synthesis algorithm. The algorithm invokes the modified bounded synthesis procedure with increasing bound k until the initial state is in B^M (environment wins) or B^m reaches a fixed point (controller wins). We prove correctness in the next section.

Algorithm 10 Amended tree formulas for Controller and Environment

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

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

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

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

  end if
end function

6.0.3 Correctness

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

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

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

$$B^M \subseteq B.$$

Correctness of SOLVEUNBOUNDED follows from these invariants. The must-invariant guarantees that the environment can force the game into

Algorithm 11 Unbounded Synthesis

```
1: function SOLVEUNBOUNDED(T)
2:     B^M ← E
3:     B^m[0] ← E
4:     for k = 1 ... do
5:         if SAT(I ∧ B^M) then return unrealisable ▷ Losing in the initial state
6:         if ∃i < k. B^m[i] ≡ B^m[i + 1] then ▷ Reached fixed point
7:             return realisable
8:         B^m[k] ← E
9:         CHECKBOUND(k)
10:    end for
11: end function
Require: May and must invariants hold
Ensure: May and must invariants hold
Ensure: I ∉ B^m[k] if there exists a winning controller strategy with bound k
Ensure: I ∈ B^M if there exists a winning environment strategy with bound k
12: function CHECKBOUND(k)
13:    return SOLVEABSTRACT(env, I, k, ∅)
14: end function
```

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

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

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

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

Line (11–12) splits the tree T into T_1 and T_2, such that T_2 has depth 1. Consider formulas $F_1 = s(X_T) \wedge \overline{\text{TREEFORMULA}}(k, T_1)$ and $F_2 = \overline{\text{TREEFORMULA}}(k, T_2)$.

These formulas only share variables X_n. Their conjunction $F_1 \wedge F_2$ is unsatisfiable, as by construction any solution of $F_1 \wedge F_2$ also satisfies $s(X_T) \wedge \overline{\text{TREEFORMULA}}(k, T)$, which is unsatisfiable (precondition (b)). Hence the interpolation operation is defined for F_1 and F_2.

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

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

Furthermore, the interpolant satisfies $F_1 \rightarrow \mathcal{I}$, i.e., any assignment to X_n that satisfies $s(X_T) \wedge \overline{\text{TREEFORMULA}}(k, T_1)$ also satisfies $\mathcal{I}$. Hence, removing $\mathcal{I}$ from $B^m[\text{HEIGHT}(k, n)]$ makes $s(X_T) \wedge \overline{\text{TREEFORMULA}}(k, T_1)$ 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, $\overline{\text{TREEFORMULA}}(k, T_1) \equiv B^m[\text{HEIGHT}(k, T_1)](X^T)$; hence $s(X_T) \wedge \overline{\text{TREEFORMULA}}(k, T_1) \equiv s(X_T) \wedge B^m[\text{HEIGHT}(k, T_1)](X^T) \equiv \bot$. Thus the second postcondition of $\overline{\text{LEARN}}$ holds.

The proof of LEARN is similar to the above proof of LEARN. An interpolant constructed from $F_1 = s(X_T) \wedge \text{TREEFORMULA}(k, T_1)$ 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: $\text{TREEFORMULA}(k, T_1) \equiv \neg B^M(X_T)$. Hence $s(X_T) \wedge \text{TREEFORMULA}(k, T_1) \equiv s(X_T) \wedge \neg B^M(X_T) \equiv \bot$. Therefore $s(X_T) \wedge B^M(X_T) \not\equiv \bot$, the second postcondition, is true.

6.0.5 Proof of Termination

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

Assume that there is a winning strategy for the controller at bound κ. This means that at some point the algorithm discovers a counterexample tree of bound κ for which the environment cannot force into E. The algorithm then invokes the $\overline{\text{LEARN}}$ method, which removes I from $B^m[\kappa]$. Alternatively, if there is a winning strategy for the environment at bound κ then a counterexample losing for the controller will be found. Subsequently LEARN will be called and I added to B^M.

6.0.6 Optimisation: 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 I assigns a value to each variable in X. If the environment loses $\langle I, k\rangle$ then we attempt to solve for a generalised version of I by removing one variable assignment at a time. If the environment loses from the larger set of states then we continue generalising. In this way we learn more states by increasing the reachable set. In our benchmarks we have observed that this optimisation is beneficial on the first few iterations of CHECKBOUND.

Algorithm 12 Generalise I optimisation

function CHECKBOUND(k)
  $r \leftarrow$ SOLVEABSTRACT($\texttt{env}, I, k, \emptyset$)
  if $r \neq \emptyset$ **then return** r
  $s' \leftarrow I$
  for $x \in X$ **do**
    $r \leftarrow$ SOLVEABSTRACT($\texttt{env}, s' \setminus \{x\}, k, \emptyset$)
    if $r =$ `NULL` **then** $s' \leftarrow s' \setminus \{x\}$ ▷ Removes the assignment to x from s'
  end for
  return `NULL`
end function

7 | Evaluation

8 Conclusion

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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