Formal Techniques for Realizability Checking and Synthesis of Infinite-State Reactive Systems

A DISSERTATION

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Andreas Katis

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Doctor of Philosophy

Dr. Michael W. Whalen and Prof. Mats P. E. Heimdahl

July, 2020

Acknowledgements

The path towards reaching this destination was long and brimming with challenges. I can humorously (w.r.t. the main subject of this thesis) say with confidence that it would have been an unrealizable goal, had I not been the lucky recipient of enormous support from my colleagues, friends and family.

First of all, I would like to thank my advisor Dr. Michael Whalen as well as Dr. Sanjai Rayardurgam and Professor Mats Heimdahl for their invaluable guidance throughout this degree. You supported my research goals in the best possible way, helped me sketch out a successful research plan, and encouraged me to explore previously unknown territories. I could simply not ask for a better environment, both in terms of professionalism, as well as day-to-day engagement. I would also like to thank the remaining members of my examination committee, Professors Steven McCamant and Marc Riedel for their service and valuable inputs.

Furthermore, I want to express my gratitude towards every other person that I collaborated with for the purposes of this work. Andrew Gacek, Arie Gurfinkel, David Greve, Darren Cofer, Grigory Fedyukovich and John Backes, thank you for the insightful discussions and ideas, as well as the immense help on writing and refining our published work. Special thanks go to the two undergraduate students that I helped mentor, Huajun Guo and Jeffrey Chen. I hope that through our collaboration I was able to spark your excitement on conducting research.

As a slight introvert, it is of utmost importance to me to be able to work in a friendly, approachable environment. As such I would like to thank my lab colleagues, and fellow graduate students in the Critical Systems (CriSys) group. Anitha, Danielle, Dongjiang, Elaheh, Hung, Ian, Jason, Kevin, Kristin, Lian, Soha and Taejoon, it has been a pleasure and a privilege to work with you in the same room. Whether it was research collaborations, or small talk, our discussions helped me stay sane and optimistic during this journey.

My thanks also go to my not-so-research-related social cycle at the University. I want to thank the Hellenic Student Association, many members of which have become dear friends of mine. Being able to share a common cultural background with other students was a catalyst towards my faster integration to the academic community, and helped ease my adjustment as an international student coming in to a completely unknown environment. I am happy that I met you all.

Moving to a different continent in order to pursue my dreams did not stop my dear friends of many years to encourage and strengthen my resolve. Δημήτρη, Νίκο, Σπύρο and Τάκη, I am forever indebted to you. You may not have realized it, but our conversations have been important for me to keep a sound mind.

My family's support played an important factor, especially given the sacrifices that they had to make in order for me to be able to reach this milestone. Many thanks go to my parents. Χριστόφορο and Καλομοίρα, and my sister Ευαγγελία, who not only allowed me, but rather reinforced my decision to pursue a higher degree.

Last but not least, I want to thank Άγη, a unique person in my life with whom I shared all of my experiences throughout these years. Your optimistic attitude, outspoken love, and formidable confidence helped me overcome all the obstacles and challenges that I had to face.

Dedication

To my parents Χριστόφορο and Καλομοίρα, my sister Γκέλυ and my friends Μήτσο, Νίκο, Σπύρο and Τάκη. But foremost, to Άγη (and the many cute nicknames that I have invented for you throughout the years).
Σας ευχαριστώ όλους.

Abstract

Reactive systems are fundamental building blocks in the development of critical safety systems. The are called "reactive" due to the necessity of them interacting with (or against) an unpredictable and uncontrollable environment, for an indefinite amount of time. As a consequence, the verification of a critical system more often than not involves the process of writing requirements, modeling, implementing and testing reactive subcomponents. Research in formal verification for reactive systems attempts to solve fundamental problems in each of these processes, with an emphasis given on the creation of mathematical proofs regarding the system's safety. More importantly, it has been shown that such proofs can lead to significant savings both in development time as well as overall production cost.

This dissertation explores the problem of *reactive synthesis*, where the goal is the development of decision procedures that are able to (dis)prove the *realizability* of reactive system specifications, as well as produce artifacts that essentially serve as witnesses to the decision. Reactive synthesis is a problem closely related to formal verification, as it requires the development of precise, mathematical proofs of realizability. The overwhelming majority of research conducted in this area has so far been focused in its application to propositional specification, where only operations over the Boolean domain can be performed. For the first part of this thesis, we propose novel, efficient reactive synthesis algorithms that extend the support for propositional specification to also generate implementations when the requirements involve the use of richer theories, such as integer and real arithmetic. We discuss the advantages and disadvantages of each algorithm, and accompany their implementation with a formal proof of soundness. In the second part of this dissertation, we delve into an unexplored sub-problem in reactive synthesis, where we present the first attempt ever to synthesize witnesses for infinite-state problems in which the implementation behaves in a random, diverse manner. While the product of synthesis is typically expected to be a deterministic witness, we argue that randomness can still be valuable in practice. To that end, we evaluate the application of randomly-behaving solutions to problems related to fuzz testing and robot motion planning, demonstrating how reactive synthesis can be used in an innovative way in software engineering concepts that have not been considered before.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software products are integral parts of today's industry, having a vast variety of applications and use cases as a means to achieve the most efficient, scalable and user-oriented solutions possible. The majority of our everyday interactions include the use of software in one way or another, and it is clear that the most subtle parts of the software's design are the ones that dictate whether the end product will meet our needs and desires. A subset of this software is particularly classified as *critical*, since it is used for applications where the end result directly affects the user's *safety* [42]. Outstanding examples of critical systems include medical devices, nuclear reactors, flight systems, as well as the emerging class of autonomous vehicles. Designing software for critical systems is hence driven with a single fundamental goal: it should never be the case that the system's behavior leads to scenarios where the user's safety is threatened. It is apparent that designing such systems is not a process to be taken lightly, but rather demands the use of sophisticated techniques that ideally provide a proof that the system under design retains the desired behavior while being safe.

For the reasons above, part of the software engineering community has focused its resources into exploring ways to reason about critical software in a transparent and well-defined manner. Being naturally industry-driven, the initial attempts delved into the development of testing processes against a finished product, where a combination of system inputs are used to examine the system behavior and compare it against a predetermined standard for a safe behavior [78]. While testing has proven to be one of the most crucial parts in software development, its fundamental weakness is due to the fact that a finished product has to first exist. In consequence, major resources are usually spent on the implementation phase, while developers make imprecise claims regarding the level of robustness of the end product, relying usually on the team's past experience. This comes in direct conflict with the overall perspective in the development of critical software, where the system's safety drives the implementation decisions, instead of being an extra feature.

This pursuit for safety brought life to work in verification, an automated process that requires means to reason about an existing implementation, and its goal is to provide a definitive answer to the developer on whether certain desired properties are met throughout the software's execution [30]. Being able to abstract away the fine details was crucial to achieving this, thus researchers focused on the creation of *models* as software "forests", but always keeping the software "tree" in perspective. Working on an abstraction makes formal reasoning and expressing properties of interest easier, reducing the problem of verification to the development of techniques that can formally prove the safety of the abstract model with respect to those properties [59]. Researchers in *formal verification* strive to take advantage of this by experimenting with the creation of logical frameworks where an abstract representation of the system is described, then verified against a set of requirements, usually described in a language flexible enough to express logical conjectures [30, 53].

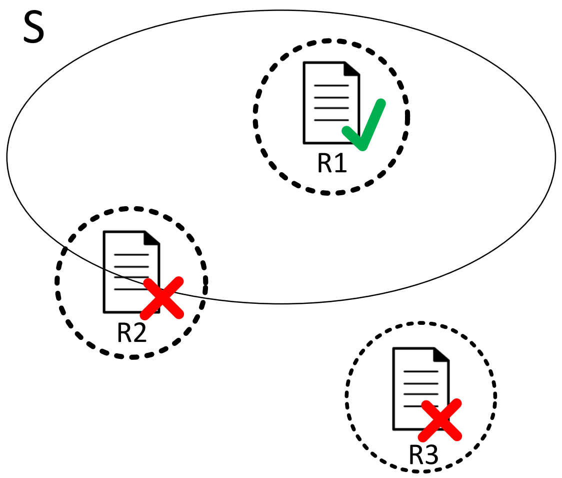


Figure 1.1: Realizable (tick) and unrealizable (cross) requirements

On a parallel track to advancements in formal verification through *model checking* [24] and *theorem proving* [40], were efforts to identify and develop automated reasoning techniques over the system requirements, prior to the existence of a final implementation [16]. The motivation was clear; a substantial amount of software failures could be traced back to erroneous or missing requirements. Formal analysis of specification was thus seen as a promising step towards the development of safer critical systems, leading not only to a higher degree of confidence provided by a proof of the specification's *consistency* [52], but also to a reduction in overall costs by avoiding the need to exhaustively debug an existing implementation. Consistency, i.e. a proof of satisfiability of the system's desired properties, is not the only desirable characteristic of a specification, though. For the purposes of this dissertation, we explore the problem of *program synthesis* [74], where we ask the following questions:

1. **Given a formal specification for the desired system, does there exist an implementation with specification-complying behavior? In other words, is the specification realizable?** Figure 1.1 provides a visual representation of

this problem. Assume that we have a state space S within which potential implementations exist that exercise some of these states. Given a set of requirements (R_1, R_2, R_3 in Figure 1.1), we want to see whether an implementation exists such that it always complies with the corresponding requirement, i.e. whether the set of requirements is realizable. This is a typical example where the answer is essentially a simple "yes" or "no", and a *decision procedure* is necessary. Nevertheless, as a useful addendum we require that the same procedure provides witnesses when the specification is *unrealizable*. For example, in Figure 1.1 specifications R_2 and R_3 are not realizable, since no implementation, i.e. no subset of states in S exists that can satisfy the corresponding requirements. This is important for the requirements engineer, as it is a minimal summary of what might be wrong with the candidate specification, and provides invaluable intuition towards not only fixing the problem, but also further understanding subtleties that cause it.

2. **Given a proof of the specification's realizability, can we define an automated procedure that will generate a witnessing program? In other words, can we synthesize an implementation?** Synthesis essentially involves the construction of a witness of the specification's realizability. The outcome is an implementation that is guaranteed by design to always satisfy constraints expressed by the engineer through the requirements, and can be viewed as a set of system behaviors that is sufficient to perform the given task. As it is minimal, a lot of implementation details are abstracted away, and it is apparent that, for the majority of its applications, it is not viewed as a candidate alternative to handwritten implementations, but it can still be used as a means to understand *what* the implementation could do, rather than *how* it should do it, to remain safe.

We further restrict the scope of this thesis to the area of *reactive systems* [57], where the system is required to indefinitely respond to external inputs over which it has no sense of control. Reactive systems are very often used as subcomponents of critical applications. Such systems have to perform in a safe manner under any allowable environment, and it is crucial to be able to reason regarding this characteristic during verification. Program synthesis in this context is usually referred to as *reactive synthesis* [86], with the goal being the examination on the specification's realizability, as well as the automated generation of a system that when run indefinitely, can safely react to its environment. Here, it is important to point out that as much effort is being put into writing requirements for the system, an equal amount of care is essential with respect to capturing its environment. Reasoning without any assumptions on the environment more often than not leads to unrealizable specification, while under- and overconstraining the environment can lead to undesirable behavior by the system, even if an implementation for it exists.

1.1 Contributions

The objective of this thesis is to propose efficient algorithms for the synthesis of reactive systems. The first key attribute that we want to achieve is to free the user from the burden of guiding the algorithms towards finding a solution, thus making the entire process completely automated. This feature comes in contrast to relevant approaches in the area, where the user has to describe the shape of the solution through the use of templates. The second contribution of this work is the introduction of a class of reactive synthesis applications, for which little to no prior work existed. We thus explore the problem of synthesizing reactive systems that exercise sets of diverse safe behaviors.

The three main research achievements presented in this dissertation are:

- a novel application of k-induction to achieve realizability checking and synthesis from the formal specification of infinite-state systems;

- a greatest-fixpoint synthesis procedure to overcome soundness issues in the k-induction algorithm;
- a synthesis procedure of systems with random behavior, which enables the application of synthesis in new, previously unexplored research problems.

The contributions are briefly discussed in the following subsections.

1.1.1 Reactive synthesis from K-Inductive Proofs of Realizability

The mathematical principle of k-induction has been used towards the formal verification of software systems [14]. The goal in this context is to construct a *k-inductive* proof that no execution trace exists, in which the system is led into a property-violating state. While standard 1-induction would be sufficient for memoryless systems, the same does not apply when its response depends on prior states. As such, a k-inductive proof identifies a least fixpoint of finite-length paths, i.e. the shortest execution path length, which is sufficient to reason about in order to determine the preservation of a system property.

While k-induction has been successfully used in the context of model checking, its usefulness with respect to the problem of program synthesis was previously unknown. In this dissertation, we present a reactive synthesis algorithm, based on the construction of a k-inductive proof of the specification's *realizability*. Realizability is a prerequisite to achieve synthesis, as an implementation can never exist for an unrealizable specification. As such, the algorithm attempts to determine whether a least fixpoint of safe states exists, such that an implementation can be synthesized. The synthesized witness is essentially an infinite traversal of states belonging in the least fixpoint.

The resulting algorithm is automated, in the sense that the user is not involved as a guide towards the successful synthesis of a witness. For the cases of unrealizable scenarios, it is capable of providing an explanation using counterexample traces where the system is eventually forced to violate at least one property. This is particularly important in terms of identifying the reasons that cause this unrealizability to manifest, and the requirements that contain conflicting statements. In addition, the technique is theory-agnostic, enabling the reasoning over specification with expressions that may involve standard arithmetic, through the use of an off-the-self Satisfiability Modulo Theories (SMT) solver.

1.1.2 Validity-Guided Reactive Synthesis

Due to the limited capabilities of state-of-the-art SMT solvers with nested quantifiers, the k-induction algorithm for realizability checking was simplified in such a way that "unrealizable" results are not sound. In other words, when the algorithm declares a specification as "unrealizable", it may be the case that this is not true, and an implementation that satisfies it can still be developed. This was the main catalyst towards discovering a technique that, while still automated and applicable to infinite-state problems, is also provably sound for both realizable and unrealizable specifications.

Towards that end, we developed a greatest fixpoint approach where the main concept is to identify the greatest set of safe system states that can be used to generate a reactive implementation, in a similar way to how Property Directed Reachability is used in verification [13]. The new algorithm is sound on both realizable and unrealizable results, and takes advantage of the Model-Based Projection technique [68] as a means to efficiently remove unsafe states through abstraction, remaining competitive in terms of performance with other infinite-state reactive synthesis tools.

1.1.3 Synthesis of Infinite-State Reactive Systems with Random Behavior

A synthesized witness is typically seen (and thought of) as a deterministic solution to the original problem. As such, given the same system state and the same stimulus provided by the system's environment, we expect to observe the same reaction by the system, no matter how many times we run the same experiment. Still, there are potential applications of reactive systems where determinism would either be insufficient or simply incompatible with the problem in question. In robot motion planning, randomness can enable the synthesis of solutions for coverage path planning [18, 50]. Furthermore, it can be considered as an additional safety measure when the robot has to perform against adversaries with learning capabilities; A random strategy is inherently harder to infer and exploit, while the current proposed techniques in active automata learning can only be used against finite-state black-box systems [60]. The idea of a reactive system with random behavior also unveils an exciting application in model-based fuzz testing, where the goal is to exhaustively test an existing system using random input sequences as test cases. In this context, given a set of requirements and input specification for the system-under-test, we can effectively synthesize an ad hoc *reactive fuzzer*, capable of utilizing information from prior tests (system coverage and crashes), to improve overall system input coverage.

In this context, we present the first approach towards synthesizing random[1] reactive systems for infinite-state problems. The idea of identifying such permissive strategies was first proposed through the Reactive Control Improvization framework [45], which currently only supports finite-state specifications. Our proposed work defines a novel Skolemization algorithm within AE-VAL, a validity detection tool for $\forall\exists$ formulas,

[1]For the sake of brevity, throughout the paper, we refer to systems that exercise random behavior using the adjective *random* (e.g. *random* system/design/witness/controller).

utilizing the concept of *uninterpreted random number generators* to capture a range of possible safe reactions, instead of deterministically opting for a specific reaction within that range. The random solutions can exhibit diverse behavior leading to bigger overall coverage of the problem's state space, and in the context of fuzz testing, function as competitive reactive fuzzers against state-of-the-art tools, in terms of both raw system coverage and capability of exposing vulnerabilities.

1.2 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2 presents the state-of-the-art in reactive synthesis, and identifies the key distinctions between prior work in the area and the techniques presented in this thesis. Chapter 3 provides the essential formal notations and definitions for this work.

Chapter 4 presents a reactive synthesis algorithm, namely JSyn, for infinite-state problems based on the construction of a k-inductive proof of realizability. First, we demonstrate how realizability checking is possible using the principle of k-induction and show how theorem proving can be used to construct machine-checked proofs of JSyn's soundness on "realizable" results. We then proceed to define a synthesis procedure which, when provided with the k-inductive proof, generates a witness reactive implementation. As the product of a Skolemization procedure over a mathematically proven proof, the witness is specification compliant by construction.

Chapter 5 presents the second synthesis algorithm, namely JSyn-vg, for which the objective was to identify a realizability checking decision procedure that provides sound answers, both for realizable and unrealizable contracts. In comparison to JSyn, JSyn-vg uses a greatest-fixpoint approach, capable of identifying an inductive set of constraints that uniquely identifies the states that can be used towards the synthesis of a safe witness.

Chapter 6 introduces the concept of synthesis of random reactive systems, and proposes a novel Skolemization algorithm in order to achieve diversity of behaviors within a synthesized witness. The algorithm is supplemented by two case studies on applications which can benefit from the use of a random reactive system.

Finally, Chapter 7 concludes this dissertation by providing a summary over its contributions, as well as an extensive discussion on future research directions.

Chapter 2

Related Work

Program synthesis is also known as “Church’s problem”, since it was first formally described by Alonzo Church in 1963 [19]. In the 1970s, Manna and Waldinger [74] first introduced a synthesis procedure using principles of theorem proving. Almost two decades later, Pnueli and Rosner [86] first formally described the implementability of reactive systems, considering first order logic formulas that stem from temporal specifications. In the same work, they provided a complete approach to synthesize finite-state implementations through the construction of deterministic Rabin automata [98].

In the recent years, program synthesis has enjoyed a vast variety of contributions under numerous contexts. Gulwani [51] presented an extended survey, hinting future research directions. Synthesis algorithms have been proposed for simple LTL specification [11, 99] subsets of it [17, 32, 67], as well as under other temporal logics [56, 76], such as SIS [4]. Chatterjee and Henzinger [15] proposed a novel component-based approach using the notion of Assume-Guarantee contracts.

Inductive synthesis is an active area of research where the main goal is the generation of an inductive invariant that can be used to describe the space of programs that are guaranteed to satisfy the given specification [41]. This idea is mainly supported by the use of SMT solvers to guide the invariant refinement through traces that violate the requirements, known as counterexamples. Our approach differentiates from this approach by only considering the capability of constructing k-inductive proofs, with no further refinement of the problem space.

Template-based approaches to synthesis described in [6, 95] focus on the exploration of programs that satisfy a specification that is refined after each iteration, following the basic principles of deductive synthesis. In particular, the E-HSF engine used in Con-Synth [6], uses a predefined set of templates to search for potential Skolem relations and thus to solve ∀∃-formulas. In contrast, our synthesis algorithms are template-free. In addition, enumeration techniques such as the one used in E-HSF is not an optimal strategy for our class of problems, since the witnesses constructed for the most complex contracts are described by nested if-then-else expressions of depth (i.e. number of branches) 10-20, a point at which space explosion is difficult to handle since the number of candidate solutions is large.

A rather important contribution in the area is the recently published work by Ryzhyk and Walker [92], where they share their experience in developing and using a reactive synthesis tool called Termite for device drivers in an industrial environment. The driver synthesis uses a predicate abstraction technique [102] to efficiently cover the state space for both safety and liveness GR(1) specifications, leveraging the theory of fixed-size vectors. The authors adopt a user-guided approach, that continuously interacts with the user in order to combat ambiguities in the specification. In contrast, our approach supports safety specifications using infinite-state, linear-arithmetic domains and follows a "hands-off", automated process.

Iterative strengthening of candidate formulas is also used in abductive inference [28] of loop invariants. Their approach generates candidate invariants as maximum universal subsets (MUS) of quantifier-free formulas of the form $\phi \Rightarrow \psi$. While a MUS may be sufficient to prove validity, it may also mislead the invariant search, so the authors use a backtracking procedure that discovers new subsets while avoiding spurious results. By comparison, in our approach the regions of validity are maximal and therefore backtracking is not required. More importantly, reactive synthesis requires mixed-quantifier formulas, and it requires that inputs are unconstrained (other than by the contract assumptions), so substantial modifications to the MUS algorithm would be necessary to apply the approach of [28] for reactive synthesis.

The concept of synthesizing implementations by discovering fixpoints was mostly inspired by the IC3 algorithm, otherwise known as Property Directed Reachability [13, 31], which was first introduced in the context of verification. Work from Cimatti *et al.* effectively applied this idea for the parameter synthesis in the HyComp model checker [20, 21]. Discovering fixpoints to synthesize reactive designs was first extensively covered by Piterman *et al.* [85] who proved that the problem can be solved in cubic time for the class of GR(1) specifications. The algorithm requires the discovery of least fixpoints for the state variables, each one covering a greatest fixpoint of the input variables. If the specification is realizable, the entirety of the input space is covered by the greatest fixpoints. In contrast, our approach computes a single greatest fixpoint over the system's outputs while avoiding any overconstraining of the input space. As the tools use different notations and support different logical fragments, practical comparisons are not straightforward.

More recently, Preiner *et al.* presented work on model synthesis [87], that employs a counterexample-guided refinement process [90] to construct and check candidate models. Internally, it relies on enumerative learning, a syntax-based technique that enumerates expressions, checks their validity against ground test cases, and proceeds to generalize the expressions by constructing larger ones. In contrast, our approach is syntax-insensitive in terms of generating regions of validity.

As the most relevant work to this thesis, Neider and Markgraf recently proposed DT-Synth, a reactive synthesis algorithm that takes advantage of active automata learning strategies, where a "teacher" and a "learner" work cooperatively towards synthesizing a winning strategy, repetitively refining it with respect to erroneous or incomplete hypotheses [79]. In comparison to DT-Synth, we present two approaches that rely on fundamentally different principles (k-indunction, fixpoint computation), and, in the case of JSyn-vg, show how our validity-guided approach leads to competitive performance.

Synthesizing reactive implementations with random behavior has only recently been expressed formally through the Control Improvisation framework [29, 44, 45], where Daniel Fremont et al. present a theoretical approach to synthesizing randomly-behaving reactive controllers from specifications that contain traditional "hard" (typically safety) properties, in addition to a set of "soft" requirements which the controller does not need to satisfy at all times. The randomness introduced is particularly desirable in applications such as robot motion planning, as well as the synthesis of reactive fuzzers for efficient test generation and, consequently, improved coverage of the system under test. The authors of the framework thus far have limited their approach to solving finite games. On the contrary in our work, we introduce a practical synthesis framework for reactive fuzzers that allows the use of specification that admits infinite theories.

Chapter 3

Formal Background

3.1 Reactive systems and Two-player Games

A common way to formally describe a reactive system is using the mathematical notion of an infinitely long *two-player game*, where each player strives to achieve an objective with respect to uncontrollable input provided by the opponent. Similarly, a critical reactive system has to indefinitely respond to its unpredictable environment, while satisfying a predefined set of safety properties. In the context of this proposal, we discuss various concepts using a variation of the minimum-backlog problem, the two player game between Cinderella and her wicked Stepmother, first expressed by Bodlaender *et al.* [9].

The main objective for Cinderella (i.e. the reactive system) is to prevent a collection of buckets from overflowing with water. On the other hand, Cinderella's Stepmother (i.e. the system's environment) refills the buckets with a predefined amount of water that is distributed in a random fashion between the buckets. For the running example, we chose an instance of the game that has been previously used in template-based synthesis [6]. In this instance, the game is described using five buckets that are placed around in a circle and each bucket can contain up to two units of water. Cinderella has the option

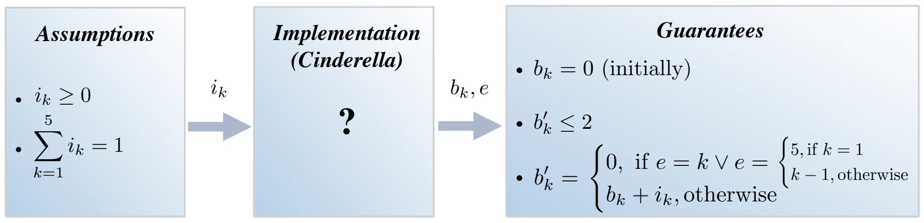

Assumptions
$i_k \geq 0$
$\sum_{k=1}^{5} i_k = 1$
i_k
Implementation (Cinderella)
?
b_k, e
Guarantees
$b_k = 0$ (initially)
$b'_k \leq 2$
$b'_k = \begin{cases} 0, & \text{if } e = k \vee e = \begin{cases} 5, \text{if } k = 1 \\ k-1, \text{otherwise} \end{cases} \\ b_k + i_k, & \text{otherwise} \end{cases}$

Figure 3.1: An Assume-Guarantee contract.

to empty two adjacent buckets at each of her turns, while the Stepmother distributes one unit of water over all five buckets.

3.2 Assume-Guarantee Contracts

One popular way to represent the system requirements is by using an *Assume-Guarantee Contract*. The *assumptions* of the contract restrict the possible inputs that the environment can provide to the system, while the *guarantees* describe safe reactions of the system to the outside world.

A (conceptually) simple example is shown in Figure 5.1. The contract describes a possible set of requirements for a specific instance of the Cinderella-Stepmother game. Our goal is to synthesize an implementation that describes Cinderella's winning region of the game. Cinderella in this case is the implementation, as shown by the middle box in Figure 5.1. Cinderella's inputs are five different values i_k, $1 \leq k \leq 5$, determined by a random distribution of one unit of water by the Stepmother. During each of her turns Cinderella has to make a choice denoted by the output variable e, such that the buckets b_k do not overflow during the next action of her Stepmother. We define the contract using the set of assumptions A (left box in Figure 5.1) and the guarantee constraints G (right box in Figure 5.1). For the particular example, it is possible to construct at least one implementation that satisfies G given A. The proof of existence of such an implementation is the main concept behind the *realizability* problem, while the automated construction of a witness implementation is the main focus of *program synthesis*.

Given a proof of realizability of the contract in Figure 5.1, we are seeking an efficient synthesis procedure that could provide an implementation. On the other hand, consider a variation of the example, where $A = true$. This is a practical case of an *unrealizable* contract, as there is no feasible Cinderella implementation that can correctly react to Stepmother's actions. A possible counterexample to realizability allows the Stepmother to pour random amounts of water into the buckets, leading to overflow of at least one bucket during each of her turns.

3.3 Formal Semantics

We use state to represent both internal state and external outputs. A transition system is a pair (I, T) where $I : \mathsf{state} \to \mathsf{bool}$ holds on the initial states states and $T : \mathsf{state} \times \mathsf{input} \times \mathsf{state} \to \mathsf{bool}$ holds on $T(s, i, s')$ when the system can transition from state s to state s' on receipt of input i. A path in this context is a sequence of transitions starting from an initial state, where each transition results to a new state that is compatible with respect to the defined transition relation.

A contract specifies the desired behavior of a transition system. A contract is a pair (A, G) of an assumption and a guarantee. The assumption $A : \mathsf{state} \times \mathsf{input} \to \mathsf{bool}$ specifies for a given system state which inputs are valid. The guarantee G is a pair (G_I, G_T) of an initial guarantee and a transitional guarantee. The initial guarantee $G_I : \mathsf{state} \to \mathsf{bool}$ specifies which states the system may start in, that is, the possible initial internal state and external outputs. The transitional guarantee $G_T : \mathsf{state} \times \mathsf{input} \times \mathsf{state} \to \mathsf{bool}$ specifies for a given state and input what states the system may transition to.

We now define what it means for a transition system to realize a contract. This requires that the system respects the guarantee for inputs which meet the assumptions. Moreover, the system must always remain responsive with respect to these inputs. In order to make this definition precise, we first need to define which system states are reachable given some assumptions on the system inputs.

Definition 3.3.1 (Reachable with respect to assumptions)**.** Let (I, T) be a transition system and let $A : \mathsf{state} \times \mathsf{input} \to \mathsf{bool}$ be an assumption. A state of (I, T) is reachable with respect to A if there exists a path starting in an initial state and eventually reaching s such that all transitions satisfy the assumptions. Formally, $\mathsf{Reachable}_A(s)$ is defined inductively by

$$\mathsf{Reachable}_A(s) = I(s) \vee \exists s_{\mathsf{prev}}, i.\ \mathsf{Reachable}_A(s_{\mathsf{prev}}) \wedge A(s_{\mathsf{prev}}, i) \wedge T(s_{\mathsf{prev}}, i, s)$$

Definition 3.3.2 (Realization)**.** A transition system (I, T) is a realization of the contract $(A, (G_I, G_T))$ when the following conditions hold

1. $\forall s.\ I(s) \Rightarrow G_I(s)$
2. $\forall s, i, s'.\ \mathsf{Reachable}_A(s) \wedge A(s, i) \wedge T(s, i, s') \Rightarrow G_T(s, i, s')$
3. $\exists s.\ I(s)$
4. $\forall s, i.\ \mathsf{Reachable}_A(s) \wedge A(s, i) \Rightarrow \exists s'.\ T(s, i, s')$

The first two conditions in Definition 3.3.2 ensure that the transition system respects the guarantees. The second two conditions ensure that the system is non-trivial and responsive to all valid inputs.

Definition 3.3.3 (Realizable)**.** A contract is realizable if there exists a transition system which is a realization of the contract.

Definitions 3.3.2 and 3.3.3 are useful for directly defining realizability, but not very useful for checking realizability. We now develop an equivalent notion which is more suggestive and amenable to checking. This is based on a notion called *viability*. Intuitively, a state is viable with respect to a contract if being in that state does not doom a realization to failure. We can capture this notion without reference to any specific realization, because condition 2 in the definition of realization tells us that G_T is an over-approximation of any T.

Definition 3.3.4 (Viable)**.** A state s is *viable* with respect to a contract $(A, (G_I, G_T))$, written $\mathsf{Viable}(s)$, if G_T can keep responding to valid inputs forever, starting from s. Informally, one can say that a state s is viable if it satisfies the infinite formula:

$$\forall i_1.\ A(s, i_1) \Rightarrow \exists s_1.\ G_T(s, i_1, s_1) \wedge \forall i_2.\ A(s_1, i_2) \Rightarrow \exists s_2.\ G_T(s_1, i_2, s_2) \wedge \forall i_3.\ \cdots$$

Formally, viability is defined coinductively by the following equation

$$\mathsf{Viable}(s) = \forall i.\ A(s, i) \Rightarrow \exists s'.\ G_T(s, i, s') \wedge \mathsf{Viable}(s')$$

Theorem 3.3.1 (Alternative realizability)**.** *A contract $(A, (G_I, G_T))$ is realizable if and only if $\exists s.\ G_I(s) \wedge \mathsf{Viable}(s)$.*

Proof. For the "only if" direction the key lemma is $\forall s.\ \mathsf{Reachable}_A(s) \Rightarrow \mathsf{Viable}(s)$. This lemma is proved by coinduction and follows directly from conditions 2 and 4 of Definition 3.3.2. Then by conditions 1 and 3 we have some state s such that $I(s)$ and $G_I(s)$. Thus $\mathsf{Reachable}_A(s)$ holds and applying the lemma we get $G_I(s) \wedge \mathsf{Viable}(s)$.

For the "if" direction, let s_0 be such that $G_I(s_0)$ and $\mathsf{Viable}(s_0)$. Define $I(s) = (s = s_0)$ and $T(s, i, s') = G_T(s, i, s') \wedge \mathsf{Viable}(s')$. Conditions 1, 2, and 3 of Definition 3.3.2

are clearly satisfied. Condition 4 follows from the observation that $\forall s.\ \mathsf{Reachable}_A(s) \Rightarrow \mathsf{Viable}(s)$ and from the definition of viability. □

Assuming that a contract is realizable with respect to Definition 3.3.1, we can then describe the program synthesis problem using the following definition.

Definition 3.3.5 (Synthesis from realizable contracts)**.** Determine an initial state s_i and function $f(s, i)$ such that $G_I(s_i)$ and $\forall s, i.\mathsf{Viable}(s) \Rightarrow \mathsf{Viable}(f(s, i))$.

We will use this definition in future chapters to describe the correctness of different synthesis algorithms.

Chapter 4

Synthesis from K-Inductive Proofs of Realizability

Our first synthesis algorithm is based on the idea of constructing *k-inductive* proofs of correctness, a prominent technique that has been successfully used in formal verification. In the context of synthesis, a proof of the specification's correctness is formally replaced by the notion of *realizability*. In Section 4.1 we present two different formal approaches to determine the realizability of a given specification and in Section 4.2 we propose a sound algorithm to synthesize implementations given a k-inductive proof of realizability.

4.1 Realizability Checking using K-Induction

In this section we describe two versions of an algorithm for automatically checking the realizability of a contract. The first version is based on Theorem 3.3.1 together with under- and over-approximations of viability. An over-approximation is useful to show that a contract is not viable, while an under-approximation is useful to show that a contract is viable. The algorithm itself is intractable, and as such we present an alternative procedure that mitigates the difficulty that solvers would otherwise face when dealing with the original formulas.

We first define an over-approximation of viability called *finite viability* based on a finite unrolling of the definition of viability. Because this is an over-approximation, if a contract does not have an initial state which is finitely viable, then the contract is not viable. We formalize this when we prove the correctness of the realizability algorithm.

Definition 4.1.1 (Finite viability). A state s is viable for n steps, written $\mathsf{Viable}_n(s)$ if G_T can keep responding to valid inputs for at least n steps. That is,

$$\begin{aligned}\mathsf{Viable}_n(s) \triangleq \forall i_1.\ A(s,i_1) \Rightarrow \exists s_1.\ G_T(s,i_1,s_1)\wedge \\ \forall i_2.\ A(s_1,i_2) \Rightarrow \exists s_2.\ G_T(s_1,i_2,s_2) \wedge \cdots \wedge \\ \forall i_n.\ A(s_{n-1},i_n) \Rightarrow \exists s_n.\ G_T(s_{n-1},i_n,s_n)\end{aligned}$$

All states are viable for 0 steps.

We next define an under-approximation of viability based on *one-step extension*. This notion looks if G_T can respond to valid inputs given a finite historical trace of valid inputs and states.

Definition 4.1.2 (One-step extension). A state s is extendable after n steps, written $\mathsf{Extend}_n(s)$, if any valid path of length n from s can be extended in response to any input. That is,

$$\begin{aligned}\mathit{Extend}_n(s) \triangleq \forall i_1, s_1, \ldots, i_n, s_n. \\ A(s,i_1) \wedge G_T(s,i_1,s_1) \wedge \cdots \wedge A(s_{n-1},i_n) \wedge G_T(s_{n-1},i_n,s_n) \implies \\ \forall i.\ A(s_n,i) \implies \exists s'.\ G_T(s_n,i,s')\end{aligned}$$

We now use these two notions to formally define our realizability algorithm. The core of the algorithm is based on two checks called the *base* and *extend* check.

Definition 4.1.3 (Realizability Algorithm)**.** Define the checks:

$$\mathsf{BaseCheck}(n) \triangleq \exists s.\ G_I(s) \wedge \mathsf{Viable}_n(s)$$

$$\mathsf{ExtendCheck}(n) \triangleq \forall s.\ \mathsf{Extend}_n(s)$$

The following algorithm checks for realizability or unrealizability of a contract.

```
for $n = 0$ to $\infty$ do
    if $\neg\mathsf{BaseCheck}(n)$ then
        return UNREALIZABLE;
    else if $\mathsf{ExtendCheck}(n)$ then
        return REALIZABLE;
end
```

Theorem 4.1.1 (Soundness of "unrealizable" result)**.** *If* $\exists n.\ \neg\mathsf{BaseCheck}(n)$ *then the contract is not realizable.*

Proof. By induction on n we prove $\forall s, n.\ \mathsf{Viable}(s) \Rightarrow \mathsf{Viable}_n(s)$. The result then follows from Theorem 3.3.1. □

Theorem 4.1.2 (Soundness of "realizable" result)**.** *If* $\exists n.\ \mathsf{BaseCheck}(n) \wedge \mathsf{ExtendCheck}(n)$ *then contract is realizable.*

Proof. First we show how $\mathsf{Extend}_n(s)$ can be used to shift $\mathsf{Viable}_n(s)$ forward. The following is proved by induction on n.

$$\forall s, n, i.\ \mathsf{Extend}_n(s) \wedge \mathsf{Viable}_n(s) \wedge A(s, i) \Rightarrow \exists s'.\ G_T(s, i, s') \wedge \mathsf{Viable}_n(s')$$

Using this lemma we can show the following by coinduction.

$$\forall s, n.\ \mathsf{Viable}_n(s) \wedge \mathsf{ExtendCheck}(n) \Rightarrow \mathsf{Viable}(s)$$

The result then follows from Theorem 3.3.1. $\square$

Corollary 4.1.1 (Soundness of Realizability Algorithm)**.** *The Realizability Algorithm is sound.*

Due to the approximations used to define $\mathsf{BaseCheck}(n)$ and $\mathsf{ExtendCheck}(n)$, the algorithm is incomplete. The following two examples show how both realizable and unrealizable contracts may send the algorithm into an infinite loop.

Example 4.1.1 (Incompleteness of "realizable" result)**.** *Suppose the type state is integers. Consider the contract:*

$$A(s,i) = \top \qquad G_I(s) = \top \qquad G_T(s,i,s') = (s \neq 0)$$

This contract is realizable by, for example, a system that starts in state 1 *and always transitions into the same state. Yet, for all* n*,* $\mathsf{ExtendCheck}(n)$ *fails since one can take a path of length* n *which ends at state* 0*. This path cannot be extended.*

Example 4.1.2 (Incompleteness of "unrealizable" result)**.** *Suppose the type state is integers. Consider the contract:*

$$A(s,i) = \top \qquad G_I(s) = (s \geq 0) \qquad G_T(s,i,s') = (s' = s - 1 \wedge s' \geq 0)$$

This contract is not realizable since in any realization the state 0 *would be reachable, but the contract does not allow a transition from state* 0*. However,* $\mathsf{BaseCheck}(n)$ *holds for all* n *by starting in state* $s = n$*.*

Implementing this algorithm requires a way of automatically checking the formulas $\mathsf{BaseCheck}(n)$ and $\mathsf{ExtendCheck}(n)$ for validity. This can be done in an SMT-solver that supports quantifiers over the language the contract is expressed in. Checking $\mathsf{ExtendCheck}(n)$ is rather nice in this setting since it has only a single quantifier alternation. Moreover, using an incremental SMT-solver one can reuse much of the work done to check $\mathsf{ExtendCheck}(n)$ to also check $\mathsf{ExtendCheck}(n+1)$. However, $\mathsf{BaseCheck}(n)$ is problematic. First, it has $2n$ quantifier alternations which puts even small cases outside the reach of modern SMT-solvers. Second, the quantifiers make it impractical to reuse the results of $\mathsf{BaseCheck}(n)$ in checking $\mathsf{BaseCheck}(n + 1)$. Finally, due to the quantifiers, a counterexample to $\mathsf{BaseCheck}(n)$ would be difficult to relay back to the user. Thus we need a simplification of $\mathsf{BaseCheck}(n)$ in order to make our algorithm practical.

Definition 4.1.4 (Simplified base check)**.** Define a simplified base check which checks that any path of length n from an initial state can be extended one step.

$$\mathsf{BaseCheck}'(n) \triangleq \forall k < n.(\forall s.G_I(s) \implies \mathsf{Extend}_k(s))$$

First, note that this check has a single quantifier alternation. Second, this check can leverage the incremental features in an SMT-solver to use the results of $\mathsf{BaseCheck}'(n)$ in checking $\mathsf{BaseCheck}'(n+1)$. Finally, when this check fails it can return a counterexample which is a trace of a system realizing the contract for n steps, but then becoming stuck. This provides very concrete and useful feedback to system developers. The correctness of this check is captured by the following theorem.

Theorem 4.1.3 (One-way soundness of simplified base check)**.**

$$(\exists s.\ G_I(s)) \Rightarrow \forall n.\ \mathsf{BaseCheck}'(n) \Rightarrow \mathsf{BaseCheck}(n)$$

Proof. By induction on n we can easily prove that:

$$\forall s, n.\ \mathsf{Extend}_n(s) \wedge \mathsf{Viable}_n(s) \Rightarrow \mathsf{Viable}_{n+1}(s)$$

The final result follows using this and induction on n. $\square$

Thus replacing $\mathsf{BaseCheck}(n)$ in the realizability algorithm with $\mathsf{BaseCheck}'(n)$ preserves soundness of the “realizability” result. However, because the implication in Theorem 4.1.3 is only in one direction, the algorithm is no longer sound for the “unrealizable” result. That is, it may return a counterexample showing n steps of a realization of the contract that gets into a stuck state. The following example makes this point explicit.

Example 4.1.3. *Consider again Example 4.1.1 where the type state is integers and the contract is:*

$$A(s,i) = \top \qquad G_I(s) = \top \qquad G_T(s,i,s') = (s \neq 0)$$

As before, this contract is easily realizable. However, $\mathsf{BaseCheck}'(n)$ *fails for all* n *since it will consider a path starting at state* n *and transitioning* n *steps to state* 0 *where no more transitions are possible.*

4.1.1 Implementation

We have built an implementation of the realizability algorithm as an extension to JKind [47], a re-implementation of the KIND model checker [54] in Java. Our tool is called JRealizability and is packaged with JKind[1]. The model's behavior is described in the Lustre language, which is the native input language of JKind and is used as an intermediate language for the AGREE tool suite, where contracts are typically expressed

[1]Latest release as of the time of writing this dissertation is 4.3.0.

using the architecture description language AADL [25, 93].

We unroll the transition relation defined by the Lustre model into SMT problems (one for the base check and another for the extend check) which can be solved in parallel. We use the SMT-LIB Version 2 format which most modern SMT solvers support [5]. The most significant issue for SMT solvers involves quantifier support, so we use the Z3 SMT solver [27] which has good support for reasoning over quantifiers and incremental search. The tool is often able to provide an answer for models containing integer and real-valued variables very quickly (in less than a second). Because of the use of quantifiers over a range of theories, it is possible that for one of the checks, Z3 returns `unknown`; in this case, we discontinue analysis. In addition, because our realizability check is incomplete, the tool terminates analysis when either a timeout or a user-specified max unrolling depth (default: 200) is reached. In this case we are able to report how far the base check reached which may provide some confidence in the realizability of the system.

4.1.2 Case Studies on Realizability Checking

As a part on testing the algorithm in actual components, we examined three different cases: a quad-redundant flight control system, a medical infusion pump, and a simple microwave controller. In this section, we provide a brief description of each case study and summarize the results in Table 4.1 at the end of the section.

Quad-Redundant Flight Control System

We ran our realizability analysis on a Quad-Redundant Flight Control System (QFCS) for NASA's Transport Class Model (TCM) aircraft simulation. We were provided with a set of English language requirements for the QFCS components and a description of the architecture. We modeled the architecture in AADL and the component requirements as assume/guarantee contracts in AGREE. As the name suggests, the QFCS consists of four redundant Flight Control Computers (FCCs). Each FCC contains components for handling faults and computing actuator signal values. One of these components is the Output Signal Analysis and Selection component (OSAS). The OSAS component is responsible for determining the output gain for signals coming from the control laws and going to the actuators. The output signal gain is determined based on the number of other faulty FCCs or based on failures within the FCC containing the OSAS component. The OSAS component contains 17 English language requirements including the following:

OSAS-S-170 – If the local Cross Channel Data Link (CCDL) has failed, OSAS shall set the local actuator command gain to 1 (one).

OSAS-S-240 – If OSAS has been declared failed by CCDL, OSAS shall set the actuator command gain to 0 (zero).

We formalized these requirements using the following guarantees:

$$\textbf{guarantee}\colon \mathsf{ccdl\_failed} \Rightarrow (\mathsf{fcc\_gain}' = 1)$$
$$\textbf{guarantee}\colon \mathsf{osas\_failed} \Rightarrow (\mathsf{fcc\_gain}' = 0)$$

These guarantees are contradictory in the case when the local CCDL has failed and the local CCDL reports to the OSAS that the OSAS has failed. This error eluded the engineers who originally drafted the requirements as well as the engineers who formalized them. In this case, there should be an assumption that if the CCDL has failed then it will not report to the OSAS that the OSAS has failed. This was not part of the original requirements. However, AGREE's realizability analysis was able to identify the error and provide a counterexample.

Medical Device Example

Our realizability tool was also used to verify the realizability of the components in the Generic Patient Controlled Analgesia infusion pump system that was described in [77]. The controller consists of six subcomponents that were given as input for the tool to verify the requirements described inside. While five of the models were proven to be realizable, a subtly incorrect requirement definition was found in the contract for the controller's infusion manager.

GPCA-1 - The mode range of the controller shall be one of nine different modes. If the controller is in one of the first two modes the commanded flow rate shall be zero.

$$
\begin{aligned}
&\textbf{guarantee}:\\
&\quad (\mathsf{IM\_OUT.Current\_System\_Mode}' \geq 0) \wedge\\
&\quad (\mathsf{IM\_OUT.Current\_System\_Mode}' \leq 8) \wedge\\
&\quad (\mathsf{IM\_OUT.Current\_System\_Mode}' = 0 \Rightarrow\\
&\qquad \mathsf{IM\_OUT.Commanded\_Flow\_Rate}' = 0) \wedge\\
&\quad (\mathsf{IM\_OUT.Current\_System\_Mode}' = 1 \Rightarrow\\
&\qquad \mathsf{IM\_OUT.Commanded\_Flow\_Rate}' = 0)
\end{aligned}
\tag{4.1}
$$

GPCA-2 - Whenever the alarm subsystem has detected a high severity hazard, then Infusion Manager shall never infuse drug at a rate more than the specified Keep Vein Open rate.

guarantee:

$$\begin{aligned}&(\mathsf{TLM\_MODE\_IN.System\_On}' \wedge \\ &\quad \mathsf{ALARM\_IN.Highest\_Level\_Alarm}' = 3) \Rightarrow \\ &(\mathsf{IM\_OUT.Commanded\_Flow\_Rate}' = \mathsf{CONFIG\_IN.Flow\_Rate\_KVO}')\end{aligned} \tag{4.2}$$

The erroneously defined guarantee (4.2) tries to assert that the IM_OUT.Commanded_Flow_Rate to some (potentially non-zero) Flow_Rate_KVO if the alarm input is 3; however, this may occur when the IM_OUT.Current_System_Mode is computed to be zero or one, in which case the flow rate is commanded to be 0. While discovering and fixing the problem was not difficult, the error was not discovered by the regular consistency check in AGREE.

Microwave Assignment

The third case study originates from a class assignment on AGREE in a graduate-level software engineering class at the University of Minnesota. The students were organized into six teams of four members. Each team was asked to specify the control software for a simplified microwave oven in AADL using a virtual integration approach. The software was split into two subsystems: one for controlling the heating element and another for controlling the display panel, with several requirements for each subsystem. The goal was to formalize these component-level requirements and use them to prove three system-level safety requirements.

Table 4.1 shows the corresponding results for each team, named as MT1, MT2, etc. While every team but one managed to provide an implementable set of requirements for the microwave's mode controller, there were several interesting cases involving the display control component. As an example, consider the two following requirements.

Microwave-1 - While the microwave is in cooking mode, $\mathsf{seconds\_to\_cook}$ shall decrease.

Microwave-3 - When the keypad is initially enabled, if no digits are pressed, the value shall be zero.

Team 6 formalized these requirements as

$$\textbf{guarantee}: (\mathsf{cooking\_mode}' = 2) \Rightarrow (\mathsf{seconds\_to\_cook}' = \mathsf{seconds\_to\_cook} - 1)$$

$$\textbf{guarantee}: (\neg\mathsf{keypad\_enabled} \wedge \mathsf{keypad\_enabled}' \wedge \neg\mathsf{any\_digit\_pressed}') \Rightarrow (\mathsf{seconds\_to\_cook}' = 0)$$

In the counterexample provided, the state where the microwave is cooking ($\mathsf{cooking\_mode} = 2$) and no digit is pressed creates a conflict regarding which value is assigned to the $\mathsf{seconds\_to\_cook}$ variable: should it decrease by one, or be assigned to zero? This counterexample is interesting because it indicates a missing assumption on the environment: the keypad is not enabled when the cooking mode is 2 (cooking). Without this assumption about the inputs, the guarantees are not realizable.

Table 4.1 contains the exact results that were obtained during the three case studies. Every "realizable" result was determined to be correct since an implementation was produced for each of the components analyzed, ensuring the accuracy of the tool. Every contract that was identified as "unrealizable" was manually confirmed to be unrealizable, i.e., there were no spurious results. Additionally, the number of steps that the base check required to provide a final answer was not more than one, with the unknown results being particularly interesting, as the tool timed out before the solver was able to provide a concrete answer. This shows that there is still work to be done in terms of the algorithm's

Case study	**Model**	**Result**	**Time elapsed (seconds)**	**Base check depth (# of steps)**
QFCS	FCS	realizable	1.762	0
QFCS	FCC	unrealizable	0.981	1
GPCA	Infusion Manager	unrealizable	0.2	1
GPCA	Alarm	realizable	0.316	0
GPCA	Config	realizable	0.102	0
GPCA	OutputBus	realizable	0.201	0
GPCA	System_Status	realizable	0.203	0
GPCA	Top_Level	realizable	0.103	0
MT 1	Mode Control	realizable	0.229	0
MT 1	Display Control	unrealizable	0.207	1
MT 2	Mode Control	realizable	0.202	0
MT 2	Display Control	unknown	1000 (tool timeout)	1
MT 3	Mode Control	realizable	0.203	0
MT 3	Display Control	unrealizable	0.202	1
MT 4	Mode Control	realizable	0.202	0
MT 4	Display Control	unrealizable	0.521	1
MT 5	Mode Control	unrealizable	0.1	1
MT 5	Display Control	unrealizable	0.222	1
MT 6	Mode Control	realizable	0.201	0
MT 6	Display Control	unknown	1000 (tool timeout)	1

Table 4.1: Realizability checking results for case studies

scalability, as well as an efficient way to eliminate quantifiers, making the solving process easier for Z3.

4.1.3 Machine-Checked Proofs for K-inductive Realizability Checking Algorithms

In the previous sections, we presented how k-induction can be used as the main engine in checking the realizability of contracts. We provided hand-proofs for several aspects of two algorithms related to the soundness of the approach with respect to both proofs and counterexamples.

Unfortunately, hand proofs of complex systems often contain errors. Given the criticality of realizability checking to our tool chain and the soundness of our computational proofs, we would like a higher level of assurance than hand proofs can provide. In this section, we provide a formalization of machine-checked proofs of correctness that ensure that the proposed k-inductive realizability algorithms will perform as expected, using the Coq proof assistant.[2] The facilities in Coq, notably mixed use of induction and co-induction, make the construction of the proofs relatively straightforward. The presented approach illustrates how interactive theorem proving and SMT solving can be used together in a profitable way; Interactive theorem proving is used for describing the soundness of the realizability checking algorithm, and the algorithm is then implemented using a SMT solver, which can automatically solve complex verification instances.

The work presented in this section is the first machine-checked formalization (to our knowledge) of a realizability checking algorithm. This is an important problem for both compositional verification involving virtual integration and component synthesis. In addition, the formalization process exposed errors regarding our initial definitions, including necessary assumptions to one of the main theorems to be proved and an error in the definition of realizability itself (we present these later in this section). While these errors did not ultimately impact the correctness of the algorithm, they underscore the importance of machine-checked proofs.

The Coq Proof Assistant

Coq[3] is an interactive tool used to formalize mathematical expressions and algorithms, and prove theorems regarding their correctness and functionality [97]. The tool was a result of the work on the calculus of constructions [26]. Its uses in the context of computer science vary, such as being a tool to represent the structure of a programming language and its characteristics, as well as to prove the correctness of underlying procedures in compilers. Compared to other mainstream interactive theorem provers, Coq is a tool

[2]The Coq file is available at https://github.com/andrewkatis/Coq/blob/master/realizability/Realizability.v

[3]The Coq Proof Assistant is available at https://coq.inria.fr/

that provides support for several aspects, such as the use of dependent types, as opposed to the Isabelle theorem prover [82], and proof by reflection, which is not supported by the PVS proof assistant [81]. A particularly essential feature is the tool's support for inductive and coinductive definitions. Definitions using the *Inductive* type in Coq represent a least fixpoint of the corresponding type and are always accompanied by an induction principle, which is implicitly used to progress through a proof by applying induction on the definition. *Coinductive* definitions, on the other hand, represent a greatest fixpoint to their type. They describe a set containing every finite or infinite instance of that type, and their uses in proofs are essentially infinite processes, built in a one-step fashion and requiring the existence of a guard condition that needs to hold for them to remain well-formed. Coinductive definitions allow a natural expression of infinite traces, which are central to our formalization of realizability, and are tedious to prove with hand-written proofs.

Definitions

The types *state* and *inputs* are used to represent a state, and a given set of inputs. We use Coq's *Prop* definition to describe the logical propositions regarding the component's *transition system* through a set I of *initial* states and the *transition* relation T between two states and a set of inputs. Finally, the contract is defined by its *assumption* and *guarantee*, with the latter being implicitly referenced by a pair of initial and transitional guarantees (*iguarantee* and *tguarantee*). The corresponding definitions in Coq are shown below. Note that we do not expect that a contract would be defined over all variables in the transition system – rather its outputs – but we do not make any distinction between internal state variables and outputs in the formalism. This way, we can use state variables to, in some cases, simplify statements of guarantees.

■ `Inductive` *inputs* `: Type :=`

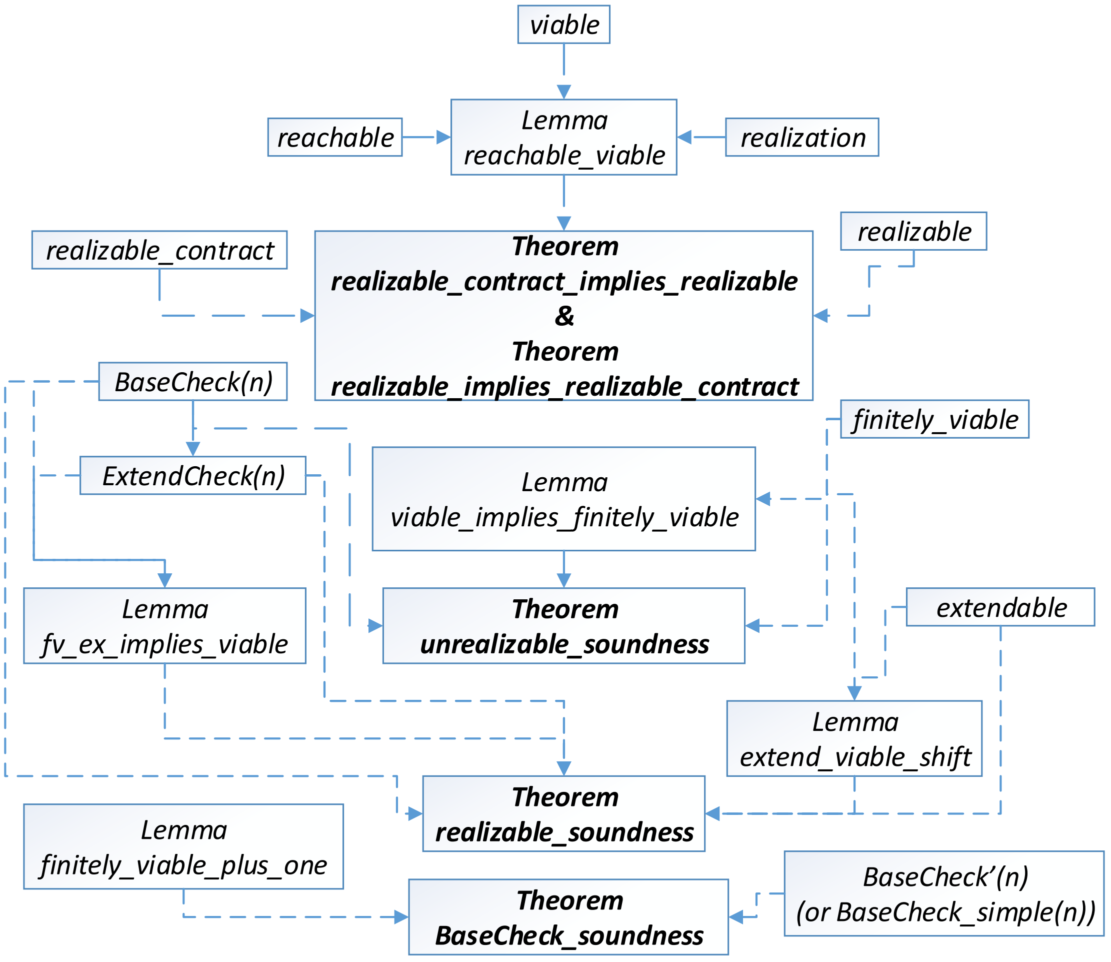

viable
reachable
Lemma reachable_viable
realization
realizable_contract
Theorem realizable_contract_implies_realizable & Theorem realizable_implies_realizable_contract
realizable
BaseCheck(n)
finitely_viable
ExtendCheck(n)
Lemma viable_implies_finitely_viable
Lemma fv_ex_implies_viable
Theorem unrealizable_soundness
extendable
Lemma extend_viable_shift
Theorem realizable_soundness
Lemma finitely_viable_plus_one
Theorem BaseCheck_soundness
BaseCheck'(n) (or BaseCheck_simple(n))

Figure 4.1: Proof Graph

$input\ :\ id\ \rightarrow\ nat\ \rightarrow\ inputs.$

- `Inductive` $state$: `Type` :=

  $st\ :\ id\ \rightarrow\ nat\ \rightarrow\ state.$

- `Definition` $initial$:= $state \rightarrow$ `Prop`.

- `Definition` $transition$:= $state \rightarrow inputs \rightarrow state \rightarrow$ `Prop`.

- `Definition` $iguarantee$:= $state \rightarrow$ `Prop`.

- `Definition` *tguarantee* := *state* → *inputs* → *state* → `Prop`.
- `Definition` *assumption* := *state* → *inputs* → `Prop`.

A state s is *reachable* with respect to the given assumptions if there exists a path from an initial state to s, while each transition in the path is satisfying the assumptions. Given a contract $(A, (G_I, G_T))$, a transition system (I, T) is its *realization* if the following four conditions hold:

1. $\forall s.\ I(s) \Rightarrow G_I(s)$
2. $\forall s, i, s'.\ reachable_A(s) \wedge A(s,i) \wedge T(s,i,s') \Rightarrow G_T(s,i,s')$
3. $\exists s.\ I(s)$
4. $\forall s, i.\ reachable_A(s) \wedge A(s,i) \Rightarrow \exists s'.\ T(s,i,s')$

Finally, we define that a given contract is *realizable*, if the existence of a transition system, which is a realization of the contract, is proved. The formalized definitions in Coq for the *reachable* state, the *realization* of a contract and whether it is *realizable* follow.

- `Inductive` *reachable* (*s* : *state*) (*I* : *initial*) (*T* : *transition*) (*A* : *assumption*) : `Prop` :=

```
rch :
  ((I s) ∨
  ((∃ (s' : state) (inp : inputs),
    (reachable s' I T A) ∧ (A s' inp) ∧ (T s' inp s))) →
  reachable s I T A).
```

- `Inductive` *realization* (*I* : *initial*) (*T* : *transition*) (*A* : *assumption*) (G_I : *iguarantee*) (G_T : *tguarantee*) : `Prop` :=

real : ((∀ (*s* : *state*), (*I s*) → (G_I *s*)) ∧

(∀ (*s s'* : *state*) (*inp* : *inputs*),

((*reachable s I T A*) ∧ (*A s inp*) ∧ (*T s inp s'*)) → G_T *s inp s'*) ∧

(∃ (*s* : *state*), *I s*) ∧

(∀ (*s* : *state*) (*inp* : *inputs*), (*reachable s I T A* ∧ (*A s inp*)) →

(∃ (*s'* : *state*), *T s inp s'*))) →

realization I T A G_I G_T.

- `Inductive` *realizable_contract* (*A* : *assumption*) (G_I : *iguarantee*) (G_T : *tguarantee*) : `Prop` :=

  rc : (∃ (*I* : *initial*) (*T* : *transition*), *realization I T A* G_I G_T) →

  realizable_contract A G_I G_T.

While the definitions of *realization* and *realizable_contract* are quite straightforward, they cannot be used directly to construct an actual realizability checking algorithm. Therefore, we proposed the notion of a state being *viable* with respect to a contract, meaning that the transition system continues to be a realization of the contract, while we are at such a state. In other words, a state is *viable* (*viable*(s)) if the transitional guarantee G_T infinitely holds, given valid inputs. Using the definition of *viable*, a contract is *realizable* if and only if $\exists s.\ G_I(s) \wedge viable(s)$.

- `CoInductive` *viable* (*s* : *state*) (*A* : *assumption*) (G_I : *iguarantee*) (G_T: *tguarantee*) : `Prop` :=

  vbl : (∀ (*inp* : *inputs*), (*A s inp*) →

  (∃ (*s'* : *state*), G_T *s inp s'* ∧ *viable s' A* G_I G_T)) →

  viable s A G_I G_T.

- `Inductive` *realizable* (*A* : *assumption*) (G_I : *iguarantee*) (G_T : *tguarantee*) : `Prop` :=

$rl : (\exists\ (s : state), G_I\ s \land viable\ s\ A\ G_I\ G_T) \rightarrow realizable\ A\ G_I\ G_T.$

Having a more useful definition for realizability, we need to prove the equivalence between the definitions of *realizable_contract* and *realizable*. The Coq definition of the theorem was split into two separate theorems, each for one of the two directions of the proof. Towards the two proofs, the auxiliary lemma that, given a realization, $\forall s.\ reachable_A(s) \Rightarrow viable(s)$ is necessary.

- `Lemma` *reachable_viable* : $\forall\ (s : state)\ (I : initial)\ (T : transition)\ (A : assumption)$ $(G_I : iguarantee)\ (G_T : tguarantee)$,

  $realization\ I\ T\ A\ G_I\ G_T \rightarrow reachable\ s\ I\ T\ A \rightarrow viable\ s\ A\ G_I\ G_T.$

The informal proof of the lemma relies initially on the unrolling of the *viable* definition, for a specific state s. Thus, we are left to prove that there exists another state s' that we can traverse into, in addition to being viable. The former can be proved directly from the conditions 2 and 4 of the definition of *realization*. For the latter, by the definition of *viable* on s' we need to show that s' is reachable. Given the definition of *reachable* though, we just need to prove that there exists another reachable state from which we can reach s', in one step. But we already know that s is such a state, and thus the lemma holds.

- `Theorem` *realizable_contract_implies_realizable* $(I : initial)\ (T : transition) : \forall\ (A : assumption)\ (G_I : iguarantee)\ (G_T : tguarantee)$,

  $realizable\_contract\ A\ G_I\ G_T \rightarrow realizable\ A\ G_I\ G_T.$

- `Theorem` *realizable_implies_realizable_contract* $(I : initial)\ (T : transition) : \forall\ (A : assumption)\ (G_I : iguarantee)\ (G_T : tguarantee)$,

  $realizable\ A\ G_I\ G_T \rightarrow realizable\_contract\ A\ G_I\ G_T.$

The first part of the theorem requires us to prove that there exists a viable state s for which the initial guarantee holds. Considering that we have a contract that is realizable under the *realizable_contract* definition, we have a transition system that is a realization of the contract, and thus from the third condition of the *realization* definition, there exists an initial state s' for which, using the first condition, the initial guarantee holds. Thus, we are left to prove that s' is viable. But, by proving that s' is reachable, we can use the *reachable_viable* lemma to show that s' is indeed viable.

The second direction requires a bit more effort. Assuming that we have a viable state s_0 with $G_I(s_0)$ being true, we define $I(s) = (s = s_0)$ and $T(s, inp, s') = G_T(s, inp, s') \wedge viable(s')$. Initially, we need to prove the *reachable_viable* lemma in this context, with the additional assumption that another viable state already exists (s_0 in this case). Having done so, we need to prove that there exists a transition system that is a realization of the given contract. Given the transition system that we defined earlier, we need to show that each of the four conditions hold. Since $I(s) = (s = s_0)$ and $G_I(s_0)$ hold, the proof for the first condition is trivial. Using the assumption that $T(s, inp, s') = G_T(s, inp, s') \wedge viable(s')$, we can also trivially prove the second condition, while the third condition is simply proved by reflexivity on the state s_0. Finally, for the fourth condition we need to prove that $\forall s, inp.\ reachable_A(s) \wedge A(s, inp) \Rightarrow \exists s'.\ G_T(s, inp, s') \wedge viable(s')$. By applying the *reachable_viable* lemma on the reachable state s in the assumptions, we show that s is also viable, if s_0 is viable, which is what we assumed in the first place. Thus, coming back into what we need to prove, and unrolling the definition of *viable* on s, we have that $\forall inp.\ A(s, inp) \Rightarrow \exists s'.\ G_T(s, inp, s') \wedge viable(s')$ which completes the proof.

Algorithms

In this section we provide a detailed description of the formalization and proof of soundness of our realizability checking algorithms within Coq. Initially, we define an under-approximation of the definition of viability, for the finite case. Recall that a state is *finitely_viable* for n steps ($viable_n(s)$), if the transitional guarantee G_T holds for at least n steps, given valid inputs.

- `Inductive` *finitely_viable* : *nat* → *state* → *assumption* → *tguarantee* → `Prop` :=

  | *fvnil* : ∀ *s A* G_T, *finitely_viable O s A* G_T

  | *fv* : ∀ *n s A* G_T, *finitely_viable n s A* G_T →

  (∀ (*inp* : *inputs*), *A s inp* → (∃ *s'*, G_T *s inp s'*)) →

  finitely_viable (*S n*) *s A* G_T.

In addition to the *finitely_viable* definition, an under-approximation of viability is also used, called one-step extension. Therefore, a valid path leading to a state s is *extendable* after n steps, if any path from s, of length at least n, can be further extended given a valid input.

- `Inductive` *extendable* : *nat* → *state* → *assumption* → *tguarantee* → `Prop` :=

  | *exnil* : ∀ (*s* : *state*) (*A* : *assumption*) (G_T : *tguarantee*),

  (∀ (*inp* : *inputs*), *A s inp* → ∃ (*s'* : *state*), G_T *s inp s'*) →

  extendable O s A G_T

  | *ex* : ∀ *n s A* G_T,

  (∀ *inp s'*, *A s inp* ∧ G_T *s inp s'* ∧ *extendable n s' A* G_T) →

  extendable (*S n*) *s A* G_T.

An Exact Algorithm for Realizability Checking

The algorithm that we propose for realizability checking consists of two checks. The *BaseCheck*(n) procedure ensures that $\exists s.\ G_I(s) \land viable_n(s)$, while *ExtendCheck*($n$) makes sure that the given state from *BaseCheck* is extendable for any n.

■ `Definition` $\mathit{BaseCheck}\ (n : \mathit{nat})\ (A : \mathit{assumption})\ (G_I : \mathit{iguarantee})\ (G_T : \mathit{tguarantee}) :=$

$\exists\ (s : \mathit{state}),\ (G_I\ s \land \mathit{finitely\_viable}\ n\ s\ A\ G_T).$

`Definition` $\mathit{ExtendCheck}\ (n : \mathit{nat})\ (A : \mathit{assumption})\ (G_T : \mathit{tguarantee}) :=$

$\forall\ s\ A\ G_T,\ \mathit{extendable}\ n\ s\ A\ G_T.$

Using the $\mathit{BaseCheck}(n)$ and $\mathit{ExtendCheck}(n)$ definitions, the algorithm determines the realizability of the given contract, using the following procedure (Section 4.1).

```
for $n = 0$ to $\infty$ do
    if $\neg$BaseCheck($n$) then
        return UNREALIZABLE
    else if ExtendCheck($n$) then
        return REALIZABLE
end
```

Using the definitions of *BaseCheck* and *ExtendCheck*, we proved the algorithm's soundness, both for the 'unrealizable' and 'realizable' case. The main idea behind the proof of soundness for the 'unrealizable' result is to prove the contrapositive, that is, given a realizable contract, there exists a natural number x for which $\mathit{BaseCheck}(x)$ holds. Unfolding the definition of $\mathit{BaseCheck}(x)$, we need to show that $\exists s.\ G_I(s) \land \mathit{viable}_x(s)$. Knowing that our assumption $\mathit{realizable\_contract}\ A\ G_I\ G_T$ is equivalent to the *realizable* definition, provides us with a state s', for which $G_I(s') \land \mathit{viable}(s')$ holds. Here, we need an additional lemma, according to which $\forall s, n.\ \mathit{viable}(s) \Rightarrow \mathit{viable}_n(s)$ (stated as $\mathit{viable\_implies\_finitely\_viable}$ below). Thus, using the lemma on $\mathit{viable}(s')$ with $n = x$, we get that $\mathit{viable}_x(s')$, thus completing the proof.

■ `Lemma` $\mathit{viable\_implies\_finitely\_viable} : \forall\ s\ A\ G_I\ G_T\ n,$

$\mathit{viable}\ s\ A\ G_I\ G_T \to \mathit{finitely\_viable}\ n\ s\ A\ G_T.$

- Theorem *unrealizable_soundness* : $\forall$ (I : *initial*) (T : *transition*) (A : *assumption*) (G_I : *iguarantee*) (G_T : *tguarantee*), ($\exists$ *n*, $\neg$*BaseCheck n A* G_I G_T) $\rightarrow$ $\neg$ *realizable_contract A* G_I G_T.

For the soundness of the 'realizable' result, we first need to prove two lemmas. Initially, *extend_viable_shift*, shows the way that $Extend_n(s)$ can be used to shift $viable_n(s)$ forward. The proof for this lemma is done by using induction on n. The base case is proved trivially, by unfolding the definitions of *extendable* and *finitely_viable* in the assumptions. For the inductive case, we assume that the same state s is extendable and finitely viable for paths of length $n + 1$, and try to prove that there exists a finitely viable state s' for paths of length $n + 1$, to which we can traverse from s, with the contract guarantees still holding after the transition. By considering that s is extendable for paths of length $n + 1$, we can use it as that potentially existing state in the proof, requiring that we can transition from s to itself, with the transitional guarantees staying true, and s being finitely viable for paths of length $n + 1$. The former is true through the definition of *extendable*, while the second is an already given assumption by the inductive step.

- Lemma *extend_viable_shift* : $\forall$ (s : *state*) (n : *nat*) (*inp* : *inputs*) (A : *assumption*) (G_I : *iguarantee*) (G_T : *tguarantee*),
  (*extendable n s A* G_T $\wedge$ *finitely_viable n s A* G_T $\wedge$ *A s inp*) $\rightarrow$
  ($\exists$ *s'*, G_T *s inp s'* $\wedge$ *finitely_viable n s' A* G_T).

- Lemma *fv_ex_implies_viable* : $\forall$ (s : *state*) (n : *nat*) (A : *assumption*) G_I G_T, (*finitely_viable n s A* G_T $\wedge$ *ExtendCheck n A* G_T) $\rightarrow$ *viable s A* G_I G_T.

- Theorem *realizable_soundness* : $\forall$ (I : *initial*) (T : *transition*) A G_I G_T, ($\exists$ *n*, (*BaseCheck n A* G_I G_T $\wedge$ *ExtendCheck n A* G_T)) $\rightarrow$ *realizable_contract A* G_I G_T.

To prove the theorem, we try to prove the equivalent for the *realizable* definition instead. The existence of a state for which the initial guarantees hold is derived from the assumption that *BaseCheck* holds for a finitely viable state, while the proof that the same state is also viable comes from the use of the $fv\_ex\_implies\_viable$ lemma, which is proved through the use of $extend\_viable\_shift$.

An Approximate Algorithm for Realizability Checking

Earlier in this section we outlined the problematic nature of *BaseCheck*(n) having $2n$ quantifier alternations, which cannot be handled efficiently by an SMT solver. To that end, we proposed a simplified version of the *BaseCheck*(n) procedure, called *BaseCheck*$'(n)$, stated as *BaseCheck_simple* below.

- `Definition` *BaseCheck_simple* (n : *nat*) (A : *assumption*) (G_I : *iguarantee*) (G_T : *tguarantee*) := $\forall$ s, (G_I s) $\rightarrow$ *extendable* n s A G_T.
- `Lemma` *finitely_viable_plus_one* : $\forall$ s n A (*gi* : *iguarantee*) (G_T : *tguarantee*) (*inp* : *inputs*),
  (*extendable* n s A G_T $\wedge$ *finitely_viable* n s A G_T) $\rightarrow$
  finitely_viable (S n) s A G_T.

- `Theorem` *BaseCheck_soundness* : $\forall$ n A (G_I : *iguarantee*) (G_T : *tguarantee*) (i : *inputs*),
  (($\exists$ s, G_I s) $\wedge$ ($\forall$ k, ($k \leq n$) $\rightarrow$ *BaseCheck_simple* k A G_I G_T)) $\rightarrow$
  BaseCheck n A G_I G_T.

The simplified *BaseCheck*$'(n)$, while being an easier instance for an SMT solver, is not sound for the 'unrealizable' case, falsely reporting some realizable contracts to not be so. Nevertheless, we proved the modified algorithm's soundness for the 'realizable' result, with the use of an auxiliary lemma.

The lemma, $finitely\_viable\_plus\_one$ simply refers to the fact that an extendable and finitely viable state s, for a given number of steps n, is also finitely viable for $n+1$ steps. The proof is done by induction on n. The base case is trivially proved, by the definition of $finitely\_viable$, and the assumption that s is extendable. For the inductive case, we use the inductive hypothesis, which leaves us to prove the assumptions on a specific state s. The extendability is trivially shown since we already know that s is extendable for paths of length $n+1$, with the same idea being applied to prove that s is finitely viable for n.

Finally, the proof of soundness for the 'realizable' result of the $BaseCheck'(n)$ procedure is done by using induction on n. The base case is trivially true, using the fact that all paths of zero length are finitely viable. The inductive step then requires us to prove that $BaseCheck(n+1)$ holds. In order to do so, we need to construct the inductive hypothesis' assumption, as a separate assumption to the theorem's scope. By applying the inductive hypothesis to the newly created assumption, we have that $BaseCheck(n)$ holds. By unrolling the definition of $BaseCheck(n)$ and applying the lemma $finitely\_viable\_plus\_one$ on the extracted state, say x, we finally prove that x is extendable through the definition of $BaseCheck'(n)$, completing the proof at the same time.

Figure 4.1 provides a simplified proof graph of all the necessary definitions and partially, for graph simplicity purposes, the way that they are used towards proving the lemmas and theorems stated in this section.

Discussion

While our work on realizability is based on simple definitions, formalizing them and refining the algorithms in Coq was non-trivial. Proving the lemmas and theorems using Coq helped us discover minor errors in our informal statements. For example, our handwritten proof of the one-way soundness Theorem 4.1.3 for the simplified $BaseCheck$ originally lacked the necessary assumption that there exists a state for which the initial guarantees hold. Another example is that we forgot to include initial states in our definition of reachable states in the informal proof. The use of a mechanized theorem prover exposed some missing knowledge in the informal text, and helped us provide a more precise version of the theorem. Although these errors in the hand proofs did not lead to problems with our implementation, Coq improved both our theorems and proofs, and provided strong assurance that our algorithm is correct.

4.2 Program Synthesis from Proofs of Realizability

The algorithm sketched in Section 4.1 can be further used for solving the more complex problem of *program synthesis* to automatically derive an implementation from the proof of a contract's realizability. Consider checks $\mathsf{BaseCheck}'(n)$ and $\mathsf{ExtendCheck}(n)$ (Definitions (4.1.4) and (4.1.3) respectively) that are used in the realizability checking algorithm. Both checks require that the reachable states explored are extendable using Definition (4.1.2). The key insights are then 1) we can start with an arbitrary state in G_I since it is non-empty, 2) we can use witnesses from the proofs of $\mathit{Extend}_k(s)$ in $\mathsf{BaseCheck}'(n)$ to create a valid path of length $n-1$, and 3) we can extend that path to arbitrary length by repeatedly using the witness of the proof of $\mathit{Extend}_n(s)$ in $\mathsf{ExtendCheck}(n)$.

In first order logic, witnesses for valid $\forall\exists$-formulas are represented by Skolem functions. Intuitively, a Skolem function expresses a connection between all universally quantified variables in the antecedents of the $\forall\exists$-formulas of $\mathsf{BaseCheck}'(n)$ and $\mathsf{ExtendCheck}(n)$, and the existentially quantified variable s' within $\mathit{Extend}_n(s)$ on the right-hand side of each implication. To generate Skolem functions from the validity of $\mathsf{BaseCheck}'(n)$ and $\mathsf{ExtendCheck}(n)$ we use the AE-VAL tool [37]. In the following subsections we present AE-VAL, and our proposed synthesis algorithm, named JSyn.

4.2.1 Witnessing existential quantifiers with AE-VAL: Validity and Skolem extraction

Skolemization is a well-known technique for removing existential quantifiers in first order formulas. Given a formula $\exists y \,.\, \psi(\vec{x}, y)$, a *Skolem function* for y, $sk_y(\vec{x})$ is a function such that $\exists y \,.\, \psi(\vec{x}, y) \iff \psi(\vec{x}, sk_y(\vec{x}))$. We generalize the definition of a Skolem function for the case of a vector of existentially quantified variables $\vec{y}$, by relaxing the relationships between elements of $\vec{x}$ and $\vec{y}$. Given a formula $\exists \vec{y} \,.\, \Psi(\vec{x}, \vec{y})$, a *Skolem relation* for $\vec{y}$ is a relation $Sk_{\vec{y}}(\vec{x}, \vec{y})$ such that 1) $Sk_{\vec{y}}(\vec{x}, \vec{y}) \implies \Psi(\vec{x}, \vec{y})$ and 2) $\exists \vec{y} \,.\, \Psi(\vec{x}, \vec{y}) \iff Sk_{\vec{y}}(\vec{x}, \vec{y})$.

The pseudocode of the AE-VAL algorithm that decides validity and extracts Skolem relation is shown in Algorithm 4.1 (we refer the reader to [37] for more detail). It assumes that the formula Ψ can be transformed into the form $\exists \vec{y} \,.\, \Psi(\vec{x}, \vec{y}) \equiv S(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, where $S(\vec{x})$ has only existential quantifiers, and $T(\vec{x}, \vec{y})$ is quantifier-free. To decide the validity, AE-VAL partitions the $\forall\exists$-formula and searches for a witnessing *local* Skolem relation of each partition. AE-VAL iteratively constructs (line 6) a set of Model-Based Projections (MBPs): $T_i(\vec{x})$, such that (a) for each i, $T_i(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, and (b) $S(\vec{x}) \implies \bigvee_i T_i(\vec{x})$. Each MBP $T_i(\vec{x})$ is connected with the local Skolem $\phi_i(\vec{x}, \vec{y})$, such that $\phi_i(\vec{x}, \vec{y}) \implies (T_{\vec{y}_i}(\vec{x}) \implies T(\vec{x}, \vec{y}))$. AE-VAL relies on an external procedure [68] to obtain MBPs for theories of Linear Real Arithmetic and Linear Integer Arithmetic.

Intuitively, each ϕ_i maps models of $S \wedge T_i$ to models of T. Thus, a *global* Skolem

Algorithm 4.1: AE-VAL$\Big(S(\vec{x}), \exists \vec{y} \,.\, T(\vec{x}, \vec{y})\Big)$, cf. [37]

```
Input: $S(\vec{x}), \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$
Output: Return value $\in \{valid, invalid\}$ of $S(\vec{x}) \Longrightarrow \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, Skolem.
Data: models $\{m_i\}$, MBPs $\{T_i(\vec{x})\}$, local Skolems $\{\phi_i(\vec{x}, \vec{y})\}$.
1  SmtAdd($S(\vec{x})$);
2  for ($i \leftarrow 0$; true; $i \leftarrow i + 1$) do
3      if (isUnsat(SmtSolve())) then
4          return valid, $Sk_{\vec{y}}(\vec{x}, \vec{y})$ from (4.3);
5      SmtPush();
6      SmtAdd($T(\vec{x}, \vec{y})$);
7      if (isUnsat(SmtSolve())) then
8          return invalid, $\varnothing$;
9      $m_i \leftarrow$ SmtGetModel();
10     $(T_i, \phi_i(\vec{x}, \vec{y})) \leftarrow$ GetMBP($\vec{y}, m_i, T(\vec{x}, \vec{y})$));
11     SmtPop();
12     SmtAdd($\neg T_i$);
13 end
```

relation $Sk_{\vec{y}}(\vec{x}, \vec{y})$ is defined through a matching of each ϕ_i against the corresponding T_i:

$$Sk_{\vec{y}}(\vec{x}, \vec{y}) \equiv \begin{cases} \phi_1(\vec{x}, \vec{y}) & \text{if } T_1(\vec{x}) \\ \phi_2(\vec{x}, \vec{y}) & \text{else if } T_2(\vec{x}) \\ \cdots & \text{else } \cdots \\ \phi_n(\vec{x}, \vec{y}) & \text{else } T_n(\vec{x}) \end{cases} \tag{4.3}$$

4.2.2 Refining Skolem relations into Skolem functions

The output of the original AE-VAL algorithm does not fulfil our program synthesis needs due to two reasons: (1) interdependencies between $\vec{y}$-variables and (2) inequalities and disequalities in the terms of the Skolem relation. Indeed, in the lower-level AE-VAL constructs each MBP iteratively for each variable $y_j \in \vec{y}$. Thus, y_j may depend on the variables of $y_{j+1}, \ldots, y_n$ that are still not eliminated in the current iteration j.

Inequalities and disequalities in a Skolem relation are not desirable because the final implementation should contain assignments to each existentially quantified variable. To specify the exact assignment value, the Skolem relation provided by AE-VAL should be post-processed to contain only equalities.

We formalize this procedure as finding a Skolem function $f_j(\vec{x})$ for each $y_j \in \vec{y}$, such that $(y_j = f_j(\vec{x})) \implies \exists y_{j+1}, \ldots, y_n \,.\, \psi_j(\vec{x}, y_j, \ldots, y_n)$. An iteration of this procedure, for some y_j, is presented in Algorithm 4.2. At the entry point, it assumes that the Skolem functions $f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$ for variables $y_{j+1}, \ldots, y_n$ are already computed. Thus, Algorithm 4.2 straightforwardly substitutes each appearance of variables $y_{j+1}, \ldots, y_n$ in ψ_j by $f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$. Once accomplished (line 4), formula $\psi_j(\vec{x}, y_j, \ldots, y_n)$ has the form $\pi_j(\vec{x}, y_j)$, i.e., it does not contain variables $y_{j+1}, \ldots, y_n$.

The remaining part of the algorithm aims to derive a function $f_j(\vec{x})$, such that $y_j = f_j(\vec{x})$. In other words, it should construct a graph of a function that is embedded in a relation. Note that AE-VAL constructs each local Skolem relation by conjoining the substitutions made in T to produce T_i. Each of those substitutions in linear arithmetic could be either an equality, inequality, or disequality. This allows us consider each $\pi_j(\vec{x}, y_j)$ to be of the following form:

$$\pi_j(\vec{x}, y_j) = L_{\pi_j} \wedge U_{\pi_j} \wedge M_{\pi_j} \wedge V_{\pi_j} \wedge E_{\pi_j} \wedge N_{\pi_j} \tag{4.4}$$

where:

$$L_{\pi_j} \triangleq \bigwedge_{l \in C(\pi_j)} (y_j > l(\vec{x})) \qquad U_{\pi_j} \triangleq \bigwedge_{u \in C(\pi_j)} (y_j < u(\vec{x})) \qquad M_{\pi_j} \triangleq \bigwedge_{l \in C(\pi_j)} (y_j \geq l(\vec{x}))$$

$$V_{\pi_j} \triangleq \bigwedge_{u \in C(\pi_j)} (y_j \leq u(\vec{x})) \qquad E_{\pi_j} \triangleq \bigwedge_{e \in C(\pi_j)} (y_j = e(\vec{x})) \qquad N_{\pi_j} \triangleq \bigwedge_{h \in C(\pi_j)} (y_j \neq h(\vec{x}))$$

Algorithm 4.2: $\textsc{ExtractSkolemFunction}(y_j, \phi(\vec{x}, \vec{y}))$

```
Input: $y_j \in \vec{y}$, local Skolem relation $\phi(\vec{x}, \vec{y}) = \bigwedge_{y_j \in \vec{y}} (\psi_j(\vec{x}, y_j, \ldots, y_n))$, Skolem
       functions $y_{j+1} = f_{j+1}(\vec{x}), \ldots, y_n = f_n(\vec{x})$
Output: Local Skolem function $y_j = f_j(\vec{x})$.
Data: Factored formula $\pi_j(\vec{x}, y_j) = L_{\pi_j} \wedge U_{\pi_j} \wedge E_{\pi_j} \wedge N_{\pi_j}$.
1  for $(i \leftarrow n; i > j; i \leftarrow i - 1)$ do
2      $\psi_j(\vec{x}, y_j, \ldots, y_n) \leftarrow \textsc{Substitute}(\psi_j(\vec{x}, y_j, \ldots, y_n), y_i, f_i(\vec{x}))$;
3  end
4  $\pi_j(\vec{x}, y_j) \leftarrow \psi_j(\vec{x}, y_j, \ldots, y_n)$;
5  if $(|E_{\pi_j}| \neq 0)$ then
6      return $E_{\pi_j}$;
7  $\pi_j(\vec{x}, y_j) \leftarrow \textsc{Merge}(L_{\pi_j}, MAX, \pi_j(\vec{x}, y_j))$;
8  $\pi_j(\vec{x}, y_j) \leftarrow \textsc{Merge}(U_{\pi_j}, MIN, \pi_j(\vec{x}, y_j))$;
9  if $(|N_{\pi_j}| = 0)$ then
10     if $(|L_{\pi_j}| \neq 0 \wedge |U_{\pi_j}| \neq 0)$ then
11         return $\textsc{Rewrite}(L_{\pi_j} \wedge U_{\pi_j}, MID, \pi_j(\vec{x}, y_j))$;
12     if $(|L_{\pi_j}| = 0)$ then
13         return $\textsc{Rewrite}(U_{\pi_j}, LT, \pi_j(\vec{x}, y_j))$;
14     if $(|U_{\pi_j}| = 0)$ then
15         return $\textsc{Rewrite}(L_{\pi_j}, GT, \pi_j(\vec{x}, y_j))$;
16 else
17     return $\textsc{Rewrite}(L_{\pi_j} \wedge U_{\pi_j} \wedge N_{\pi_j}, FMID, \pi_j(\vec{x}, y_j))$;
```

We present several primitives needed to construct $y_j = f_j(\vec{x})$ from $\pi_j(\vec{x}, y_j)$ based on the analysis of terms in L_{π_j}, U_{π_j}, M_{π_j}, V_{π_j}, E_{π_j} and N_{π_j}. For simplicity, we omit some details on dealing with non-strict inequalities in M_{π_j} and V_{π_j} since they are similar to strict inequalities in L_{π_j} and U_{π_j}. Thus, without loss of generality, we assume that M_{π_j} and V_{π_j} are empty. In the description, we denote the number of conjuncts in formula A as $|A|$. We focus on Linear Real Arithmetic in this section; and the corresponding routine for Linear Integer Arithmetic is worked out similarly.

The simplest case (line 5) is when there is at least one conjunct $(y_j = e(\vec{x})) \in E_{\pi_j}$. Then $(y_j = e(\vec{x}))$ itself is a Skolem function. Otherwise, the algorithm creates a Skolem function from the following primitives.

Definition 4.2.1. Let $l(\vec{x})$ and $u(\vec{x})$ be two terms in linear real arithmetic, then operators MAX, MIN, MID, LT, GT are defined as follows:

$$\begin{aligned} GT(l)(\vec{x}) &\triangleq l(\vec{x}) + 1 & MAX(l, u)(\vec{x}) &\triangleq ite(l(\vec{x}) < u(\vec{x}), u(\vec{x}), l(\vec{x})) \\ LT(u)(\vec{x}) &\triangleq u(\vec{x}) - 1 & MIN(l, u)(\vec{x}) &\triangleq ite(l(\vec{x}) < u(\vec{x}), l(\vec{x}), u(\vec{x})) \\ MID(l, u)(\vec{x}) &\triangleq \frac{l(\vec{x}) + u(\vec{x})}{2} \end{aligned}$$

Lemma 1. *If $|L_{\pi_j}| = n$, and $n > 1$, then L_{π_j} is equivalent to $y_j > MAX(l_1, MAX(l_2, \ldots MAX(l_{n-1}, l_n)))(\vec{x})$. If $|U_{\pi_j}| = n$, and $n > 1$, then U_{π_j} is equivalent to $y_j < MIN(u_1, MIN(u_2, \ldots MIN(u_{n-1}, u_n)))(\vec{x})$.*

This primitive is applied (lines 7-8) in order to reduce the size of L_{π_j} and U_{π_j}. Thus, from this point on, with out loss of generality, we assume that each L_{π_j} and U_{π_j} have at most one conjunct.

Lemma 2. *If $|L_{\pi_j}| = |U_{\pi_j}| = 1$, and $|E_{\pi_j}| = |N_{\pi_j}| = 0$, then the Skolem relation can be rewritten into $y_j = MID(l, u)(\vec{x})$.*

This primitive is applied (line 11) in case the graph of a Skolem function can be constructed exactly in the middle of the two graphs for the lower- and the upper boundaries for the Skolem relation. Otherwise, if some of the boundaries are missing, but $|N_{\pi_j}| = 0$ (lines 13-15), then the following primitive is applied:

Lemma 3. *If $|L_{\pi_j}| = 1$, and $|U_{\pi_j}| = |E_{\pi_j}| = |N_{\pi_j}| = 0$, then the Skolem relation can be rewritten into the form $y_j = GT(l)(\vec{x})$. If $|U_{\pi_j}| = 1$, and $|L_{\pi_j}| = |E_{\pi_j}| = |N_{\pi_j}| = 0$, then the Skolem relation can be rewritten into the form $y_j = LT(l)(\vec{x})$.*

Finally, the algorithm handles the cases when $|N_{\pi_j}| > 0$ (line 17). We introduce operator *FMID* that for given l, u, and h and each $\vec{x}$ outputs either $\mathit{MID}(l,u)$ or $\mathit{MID}(l,\mathit{MID}(l,u))$ depending on if $\mathit{MID}(l,u)$ equals to h or not.

Lemma 4. *If $|L_{\pi_j}| = |U_{\pi_j}| = |N_{\pi_j}| = 1$, and $|E_{\pi_j}| = 0$, then the Skolem relation can be rewritten into the form $y_j = \mathit{FMID}(l,u,h)(\vec{x})$, where*

$$\mathit{FMID}(l,u,h)(\vec{x}) \triangleq \mathit{ite}\Big(\mathit{MID}(l,u)(\vec{x}) = h(\vec{x}),\\ \mathit{MID}\big(l,\mathit{MID}(l,u)\big)(\vec{x}),\ \mathit{MID}(l,u)(\vec{x})\Big)$$

For $|N_{\pi_j}| > 1$, the Skolem gets rewritten in a similar way recursively.

Lemma 5. *If $|L_{\pi_j}| = |N_{\pi_j}| = 1$, and $|E_{\pi_j}| = |U_{\pi_j}| = 0$, then the Skolem relation can be rewritten into the form $y_j = \mathit{FMID}(l, \mathit{GT}(l), h)(\vec{x})$. If $|U_{\pi_j}| = |N_{\pi_j}| = 1$, and $|E_{\pi_j}| = |L_{\pi_j}| = 0$, then the Skolem relation can be rewritten into the form $y_j = \mathit{FMID}(\mathit{LT}(u), u, h)(\vec{x})$.*

Theorem 4.2.1 (Soundness)**.** *Iterative application of Algorithm 4.2 to all variables $y_n, \ldots, y_1$ returns a local Skolem function to be used in* (4.3)*.*

Proof. Follows from the case analysis that applies the lemmas above. □ □

Recall that the presented technique is designed to effectively remove inequalities and disequalities from local Skolem relations. The resulting local Skolem functions enjoy a more fine-grained and easy-to-understand form. We admit that for further simplifications (that would benefit the synthesis procedure), we can exploit techniques to rewrite *MBP*-s into compact *guards* [37].

4.2.3 Synthesis Algorithm

Algorithm 4.3, named JSyn, provides a summary of the synthesis procedure. The algorithm first determines whether the set of initial states G_I is non-empty. Second, it

Algorithm 4.3: JSyn (A : assumptions, G : guarantees)

```
Input: $A(s,i)$ : assumptions, $G_I(s), G_T(s,i,s')$ : guarantees
Output: $\langle$REALIZABLE, $Skolems\rangle$/UNREALIZABLE
1  $Skolems \leftarrow \langle\rangle$;
2  $InitResult \leftarrow$ Sat?($G_I$);
3  if (isUnsat($InitResult$)) then
4      return UNREALIZABLE;
5  for ($i \leftarrow 0$; true; $i \leftarrow i+1$) do
6      $\langle valid, Skolem\rangle \leftarrow$ AE-VAL(ExtendCheck($i$));
7      if $valid$ then
8          $Skolems.Add(Skolem)$;
9          return $\langle$REALIZABLE, $Skolems\rangle$;
10     $\langle valid, Skolem\rangle \leftarrow$ AE-VAL(BaseCheck$'_k$($i$));
11     if $\neg valid$ then
12         return UNREALIZABLE;
13     $Skolems.Add(Skolem)$;
14 end
```

attempts to construct an inductive proof of the system's realizability, using AE-VAL to find Skolem witnesses. For each call of AE-VAL we write $\langle x, y\rangle \leftarrow$ AE-VAL(...): x specifies if the given formula is valid or not, and y contains the Skolem function (only in case of the validity). The algorithm iteratively proves $\mathsf{BaseCheck}'_k(i) \triangleq \forall s.G_I(s) \implies Extend_i(s), \forall k.k < i$ and accumulates the resulting Skolem functions. If $\mathsf{BaseCheck}'_k(i)$ ever fails, we know $\mathsf{BaseCheck}'(i)$ would also fail and so the system is unrealizable. At the same time, the algorithm tries to prove $ExtendCheck(i)$. As soon as the inductive step of $ExtendCheck(i)$ passes, we have a complete k-inductive proof stating that the contract is realizable. We then complete our synthesis procedure by generating a Skolem function that corresponds to the inductive step, and return the list of the Skolem functions. Note that in Algorithm 4.3 for a particular depth k, we perform the extends check prior to the base check. The intuition is that $\mathsf{BaseCheck}'(i)$ checks $\forall k < i$; thus, it is one step "smaller" than the extends check and this avoids a special case at $k = 0$.

Given a list of Skolem functions, it remains to plug them into an implementation

Template 4.1: Structure of an implementation

```
ASSIGN_INIT();
READ_INPUTS();
SKOLEMS[0]();
...
READ_INPUTS();
SKOLEMS[$k-1$]();
while $true$ do
    READ_INPUTS();
    SKOLEMS[$k$]();
    UPDATE_HISTORY();
end
```

skeleton as shown in Template 4.1. The combination of Lustre models and k-inductive proofs allow the properties in the model to manipulate the values of variables up to $k-1$ steps in the past. Thus, the first step of an implementation (method ASSIGN_INIT()) creates an array for each state variable of size $k+1$, where k is the depth of the solution to Algorithm 4.3. This array represents the depth of history necessary to compute the recurrent Skolem function produced by the ExtendCheck(n) process. The BaseCheck$'(n)$ Skolem functions initialize this history.

In each array, the i-th element, with $0 \leq i \leq k-1$, corresponds to the value assigned to the variable after the call to i-th Skolem function. As such, the first $k-1$ elements of each array correspond to the $k-1$ Skolem functions produced by the BaseCheck$'(n)$ process, while the last element is used by the Skolem function generated from the formula corresponding to the ExtendCheck(n) process.

The template uses the Skolem functions generated by AE-VAL for each of the BaseCheck$'(n)$ instances to describe the initial behavior of the implementation prior to depth k. There are two "helper" operations: UPDATE_HISTORY() shifts each element in the arrays one position forward (the 0-th value is simply forgotten), and READ_INPUTS() reads the current values of inputs into the i-th element of the variable arrays, where i

represents the i-th step of the process. Once the history is entirely initialized using the $\mathsf{BaseCheck}'(n)$ Skolem functions, we add the Skolem function for the $\mathsf{ExtendCheck}(n)$ instance to describe the recurrent behavior of the implementation (i.e., the next value of outputs in each iteration in the infinite loop).

To establish the correctness of the algorithm, we present proofs as to the validity of $BaseCheck'(n)$ and $ExtendCheck(n)$ using the Skolem functions. The majority of the contracts are expected to have realizability proofs of length equal to 0 or 1, and as such, we limit our proofs of correctness to these two specific cases.

Theorem 4.2.2 (Bounded Soundness of BaseCheck and ExtendCheck using Skolem Functions)**.** *Let* $\mathsf{BaseCheck}_{S(s_n,i,s')}(n)$ *and* $\mathsf{ExtendCheck}_{S(s_n,i,s')}(n)$*,* $n \in 0, 1$*, be the valid variations of the corresponding formulas* $BaseCheck(n)$ *and* $ExtendCheck(n)$*, where the existentially quantified part* $\exists s'.\ G_T(s_n, i, s')$ *has been substituted with a witnessing Skolem function* $S(s_n, i, s')$*. We have that:*

- $\forall (A, G_I, G_T). BaseCheck(n) \Rightarrow BaseCheck_{S(s_n,i,s')}(n)$
- $\forall (A, G_I, G_T). ExtendCheck(n) \Rightarrow ExtendCheck_{S(s_n,i,s')}(n)$

Proof. The proof uses the definition $Extend_n(s)$ of an extendable state, after replacing the next-step states with corresponding Skolem functions. From there, the proof of the two implications is straightforward. □ □

4.2.4 An Illustrative Example

The left side of Figure 4.2 shows a (somewhat contrived) contract for a system that detects whether a string of two zeros or two ones ever occurs in a stream of inputs written in a dialect of the Lustre language [61]. The right side shows a possible implementation of that contract, visualized as a state machine. Rather than use this implementation, we

```
node top(x : int; state: int) returns (  );
var
  bias : int;
  guarantee1, guarantee2, guarantee3,
  guarantee4, guarantee5, guarantee_all : bool;
  bias_max : bool;
let
  bias = 0 -> (if x = 1 then 1 else -1) + pre(bias);
  bias_max = false ->
        (bias >= 2 or bias <= -2) or pre(bias_max);
  assert (x = 0) or (x = 1);
  guarantee1 = (state = 0 => (bias = 0));
  guarantee2 = true ->
        (pre(state = 0) and x = 1) => state = 2;
  guarantee3 = true ->
        (pre(state = 0) and x = 0) => state = 1;
  guarantee4 = bias_max => state = 3;
  guarantee5 = state = 0 or state = 1
                   or state = 2 or state = 3;
  guarantee_all = guarantee1 and guarantee2 and
          guarantee3 and guarantee4 and guarantee5;
  --%PROPERTY guarantee_all;
  --%REALIZABLE x;
tel;
```

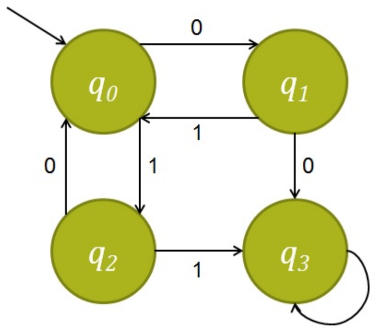


Figure 4.2: Requirements and possible implementation for example

would like to synthesize a new one directly from the contract. There are two unassigned variables in the contract, `x` and `state`. The `--%REALIZABLE` statement specifies that `x` is a system input, and by its absence, that `state` is a system output. The contract's assumption is specified by the `assert` statement and restricts the allowable input values of `x` to either 0 or 1. We also have five guarantees: `guarantee2` and `guarantee3` are used to indirectly describe some possible transitions in the automaton;[4] `guarantee5` specifies the range of values of variable `state`; `guarantee1` and `guarantee4` are the requirements with respect to two local variables, `bias` and `bias_max`, where `bias` calculates the number of successive ones or zeros read by the automaton and `bias_max` indicates that at least two zeros or two ones have been read in a row.

Note that while Lustre is a compilable language, using standard compilation tools the "program" in Figure 4.2 would not compile into a meaningful implementation: it has no outputs! Instead, it defines the guarantees we wish to enforce within the controller, and our synthesis tool will construct a program which meets the guarantees.

The realizability check on this example succeeds with a k-inductive proof of length

[4]In Lustre, the arrow (->) and `pre` operators are used to provide an initial value and access the previous value of a stream, respectively.

```
if (((x[1] == 1 && (-1 == bias[0])) || (x[1] == 0 && (1 == bias[0])))
     && !bias_max[0] && (state[0] != 0 || x[1] == 0)
     && (!state[0] != 0 || x[1] == 1)) {
  bias_max[1] = 0;
  bias[1] = 0;
  state[1] = 0;
}
```

Figure 4.3: A code snippet of the synthesized implementation for the contract from Fig. 4.2.

$k = 1$. The two corresponding $\forall\exists$-formulas ($k = 0$ for the base check and $k = 1$ for the inductive check) are valid, and thus AE-VAL extracts two witnessing Skolem functions that effectively describe assignments to the local variables of the specification, as well as to `state` (see Appendix B for the particular formulas).

The Skolem functions are used to construct the final implementation following the outline provided in Template 4.1. The main idea is to redefine each variable in the model as an array of size equal to k and to use the k-th element of each array as the corresponding output of the call to k-th Skolem function. After this initialization process, we use an infinite loop to assign new values to the element corresponding to the last Skolem function, to cover the inductive step of the original proof. The final code, a snippet of which is presented below, is 144 lines long. Since each Skolem is represented by an *ite*-statement (to be explained in Section 6.4), each branch is further encoded into a C-code, as shown in Figure 4.3.

Notice how each variable is represented by an array in the snippet above. We chose to use this easy to understand representation in order to effectively store all the past $k-1$ values of each variable, that may be needed during the construction of the k-inductive proof.

Recall that the user-defined model explicitly specifies only two transitions (via `guarantee2` and `guarantee3`), while the set of implicitly defined transitions (via `guarantee1` and `guarantee4`) is incomplete. Interestingly, our synthesized implementation turns all implicit transitions into explicit ones which makes them executable and, furthermore, adds the missing ones (e.g., as in the aforementioned snippet, from `state = 1` to `state = 0`).

4.2.5 Implementation

We developed JSYN, our synthesis algorithm on top of JKIND [47], a Java implementation of the KIND model checker.[5] Each model is described using the Lustre language, which is used as an intermediate language to formally verify contracts in the Assume-Guarantee Reasoning (AGREE) framework [25]. Internally, JKIND uses two parallel engines (for *BaseCheck* and *ExtendCheck*) in order to construct a k-inductive proof for the property of interest. The first order formulas that are being constructed are then fed to the Z3 SMT solver [27] which provides state of the art support for reasoning over quantifiers and incremental search. For all valid $\forall\exists$-formulas, JSYN proceeds to construct a list of Skolem functions using the AE-VAL Skolemizer. AE-VAL supports LRA and LIA and thus provides the Skolem relation over integers and reals.[6]

As discussed in Section 4.2, we construct a Skolem function for each base check up to depth k and one for the inductive relation at depth k. What remains is to knit those functions together into an implementation in C. The SMTLIB2C tool performs this translation, given an input list of the original Lustre specification (to determine the I/O interface) and the Skolem functions (to define the behavior of the implementation). The main translation task involves placing the Skolem functions into the template described in Template. 4.1. Each Skolem function describes a bounded history of at most depth k over specification variables, so each variable is represented by an array of size k in the generated program. The tool ensures that the array indices for history variables match

[5]An unofficial release of JKIND supporting synthesis is available to download at https://github.com/andrewkatis/jkind-1/tree/synthesis. AE-VAL needs to be installed separately from https://github.com/grigoryfedyukovich/aeval.

[6]For realizability checks over Linear Integer and Real Arithmetic (LIRA), JKIND has an option to use Z3 directly.

up properly across the successive base- and inductive-case Skolem functions. Note that during this translation process, real variables in Skolem functions are defined as floats in C, which could cause overflow and precision errors in the final implementation. We will address this issue in future work.

4.2.6 Experimental Results

For the purposes of this work, we synthesized implementations for 58 contracts written in Lustre, including the running example from Figure 4.2.[7] The original Lustre programs contained both the contract as well as an implementation, which provided us with a complete test benchmark suite since we were able to compare the synthesized implementations to already existing handwritten solutions. By extracting the handwritten implementation, we synthesized an alternative, and translated both versions to an equivalent C representation, using SMTLib2C for the synthesized programs and the LustreV6 compiler [61], including all of its optimization options, for the original implementations.

Figure 4.4 shows the overhead of synthesis by JSyn comparing to the realizability checking by JKind, while Figure 5.4a provides a scatter plot of the results of our experiments in terms of the performance of the synthesized programs against the original, handwritten implementations. Each dot in the scatter plot represents a pair of running times (x - handwritten, y - synthesized) of the 58 programs.For the two most complex models in the benchmark suite, the synthesized implementations underperform the programs generated by LustreV6. As the level of complexity decreases, we notice that both implementations share similar performance levels, and for the most trivial contracts in the experiment set, the synthesized programs perform better with a noticeable gap. We attribute these results mainly to the simplicity of the requirements expressed in the

[7]The benchmarks can be found at https://github.com/andrewkatis/synthesis-benchmarks/tree/master/verification.

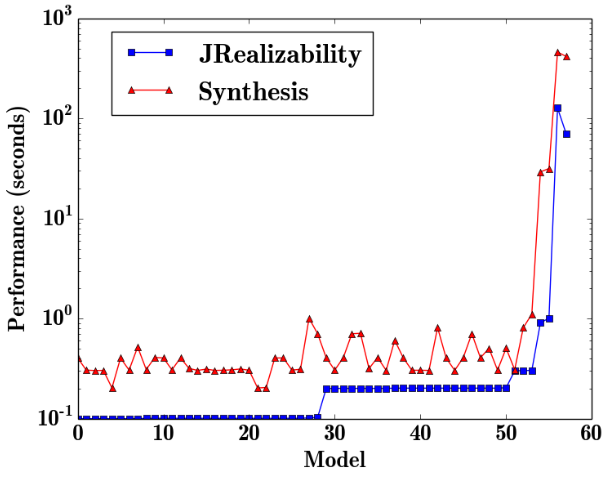


Figure 4.4: Performance comparison (time) of realizability checking and synthesis

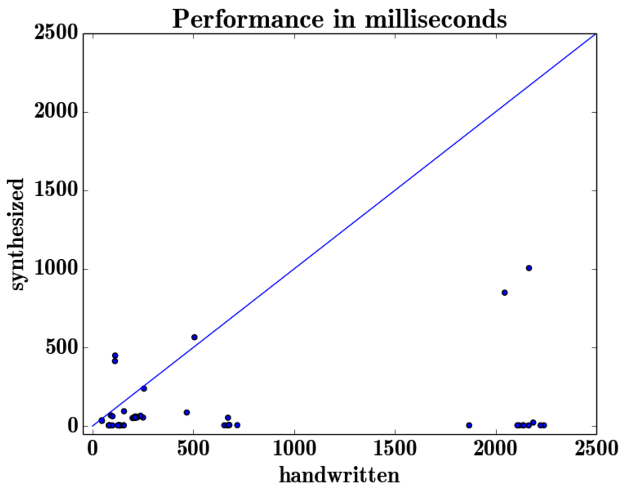


Figure 4.5: Performance comparison (time) of synthesized and handwritten implementations

majority of the models which were proved realizable for $k = 0$ by JKind, except for the example from Section 5.1 and two complex contracts for a cruise controller, which were proved for $k = 1$. It is important at this point to recall the fact that the synthesized implementations are not equivalent to the handwritten versions, in a similar fashion to the example used in Section 5.1.

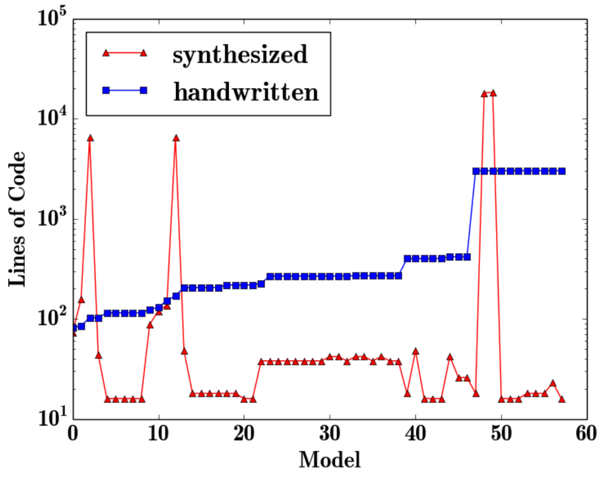


Figure 4.6: Size comparison (lines of code) of synthesized and handwritten implementations

Figure 4.6 presents the size of the implementations. Here, we can see the direct effect of the specification complexity to the size of the Skolem functions generated by AE-VAL. Two out of the five synthesized programs that are larger than their handwritten counterparts were also slower than the handwritten implementations. Since the majority of the models contained simple requirements, the overall size of the synthesized implementation remained well below LUSTREV6-programs.

Handwritten implementations are still prevalent in application domains since they provide advantages in numerous aspects, such as readability, extendability and maintenance. Nevertheless, the results show that the synthesized implementations can be used as efficient placeholders to reduce the time required to verify a system under construction, without needing a final implementation for all its components.

4.3 Conclusion

In this chapter, we presented a novel approach to program synthesis guided by the proofs of realizability of Assume-Guarantee contracts. To check realizability, it performs k-induction-based reasoning to decide validity of a set of $\forall\exists$-formulas. Whenever a contract is proven realizable, it further employs the Skolemization procedure in AE-VAL and extracts a fine-grained witness to realizability. The generated Skolem functions are then translated into an executable implementation. We implemented the technique in the JSyn tool and evaluated it for the set of Lustre models of different complexity. The experimental results provided fruitful conclusions on the overall efficacy of the the approach.

To the best of our knowledge our work is the first complete attempt at providing a synthesis algorithm based on the principle of k-induction using infinite theories. The ability to express contracts that support ideas from many categories of specifications, such as template-based and temporal properties, increases the potential applicability of this work to multiple subareas on synthesis research.

As a notable byproduct of this work, the machine-checked proofs that we developed in Coq were crucial towards verifying our approach and learning more about the actual functionality of the algorithm. Interactive theorem provers like Coq provide the necessary support to define the notions and assertions while being able to effectively prove theorems in a far more convenient and reassuring way, in contrast to hand-written, informal proofs, especially when it comes down to tracking formulas containing alternating quantifiers. Furthermore, the procedure of proving the theorems in an interactive way with a tool allowed us to refine our definitions. Additionally, the time that was required was minimal when compared to the process of considering the informal proofs and writing down our requirements in English. The most important outcome was the proof of correctness of our approach that enabled us to provide a complementary set of definitions and proofs, easily processed by an experienced Coq user.

Chapter 5

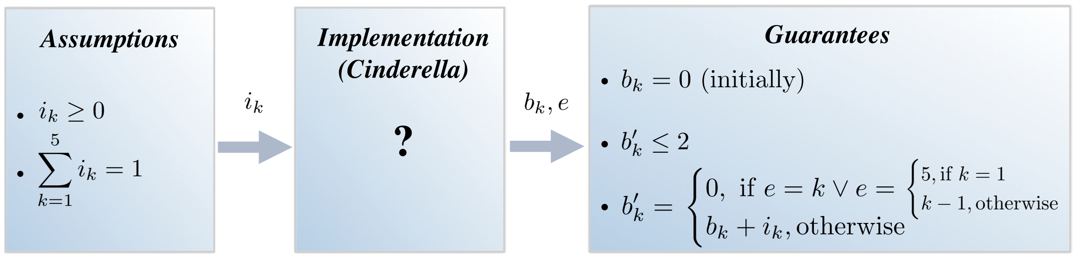
Validity-Guided Reactive Synthesis

The lack of soundness for unrealizable results in JSYN made imperative the pursuit for a better approach. The intuition behind the second algorithm in this thesis relies on the discovery of a greatest fixpoint F that only contains viable states. We can determine whether F is a fixpoint by examining the validity of the following formula:

$$\forall s, i.\ (F(s) \wedge A(s, i) \Rightarrow \exists s'.G_T(s, i, s') \wedge F(s'))$$

In the case where the greatest fixpoint F is non-empty, we check whether it satisfies G_I for some initial state. If so, we proceed by extracting a witnessing initial state and witnessing skolem function $f(s, i)$ to determine s' that is, by construction, guaranteed to satisfy the specification.

To achieve witness extraction, we again depend on AE-VAL, with its support for generating *regions of validity* being particularly crucial for the computation of fixpoints.



Assumptions
$i_k \geq 0$
$\sum_{k=1}^{5} i_k = 1$
i_k
Implementation (Cinderella)
?
b_k, e
Guarantees
$b_k = 0$ (initially)
$b'_k \leq 2$
$b'_k = \begin{cases} 0, & \text{if } e = k \vee e = \begin{cases} 5, \text{if } k = 1 \\ k-1, \text{otherwise} \end{cases} \\ b_k + i_k, & \text{otherwise} \end{cases}$

Figure 5.1: An Assume-Guarantee contract.

5.1 Overview: The Cinderella-Stepmother Game

We illustrate the flow of the validity-guided synthesis algorithm using a variation of the minimum-backlog problem, the two player game between Cinderella and her wicked Stepmother, first expressed by Bodlaender *et al.* [9].

The main objective for Cinderella (i.e. the reactive system) is to prevent a collection of buckets from overflowing with water. On the other hand, Cinderella's Stepmother (i.e. the system's environment) refills the buckets with a predefined amount of water that is distributed in a random fashion between the buckets. For the running example, we chose an instance of the game that has been previously used in template-based synthesis [6]. In this instance, the game is described using five buckets, where each bucket can contain up to two units of water. Cinderella has the option to empty two adjacent buckets at each of her turns, while the Stepmother distributes one unit of water over all five buckets. In the context of this chapter we use this example to show how specification is expressed, as well as how we can synthesize an efficient implementation that describes reactions for Cinderella, such that a bucket overflow is always prevented.

We represent the system requirements using an Assume-Guarantee Contract. The assumptions of the contract restrict the possible inputs that the environment can provide to the system, while the guarantees describe safe reactions of the system to the outside world.

A (conceptually) simple example is shown in Figure 5.1. The contract describes a possible set of requirements for a specific instance of the Cinderella-Stepmother game. Our goal is to synthesize an implementation that describes Cinderella's winning region of the game. Cinderella in this case is the implementation, as shown by the middle box in Figure 5.1. Cinderella's inputs are five different values i_k, $1 \leq k \leq 5$, determined by a random distribution of one unit of water by the Stepmother. During each of her turns Cinderella has to make a choice denoted by the output variable e, such that the buckets b_k do not overflow during the next action of her Stepmother. We define the contract using the set of assumptions A (left box in Figure 5.1) and the guarantee constraints G (right box in Figure 5.1). For the particular example, it is possible to construct at least one implementation that satisfies G given A which is described in Section 5.3.3. The proof of existence of such an implementation is the main concept behind the *realizability* problem, while the automated construction of a witness implementation is the main focus of *program synthesis*.

Given a proof of realizability of the contract in Figure 5.1, we are seeking an efficient synthesis procedure that could provide an implementation. On the other hand, consider a variation of the example, where $A = \mathit{true}$. This is a practical case of an *unrealizable* contract, as there is no feasible Cinderella implementation that can correctly react to Stepmother's actions. An possible counterexample allows the Stepmother to pour random amounts of water into the buckets, leading to overflow of at least one bucket during each of her turns.

5.2 Skolem functions and regions of validity

In order to decide the validity of $\forall\exists$-formulas and extract Skolem functions, we rely on the already established algorithm called AE-VAL. It takes as input a formula $\forall x \,.\, \exists y \,.\, \Phi(x, y)$ where $\Phi(x, y)$ is quantifier-free. To decide its validity, AE-VAL first normalizes $\Phi(x, y)$

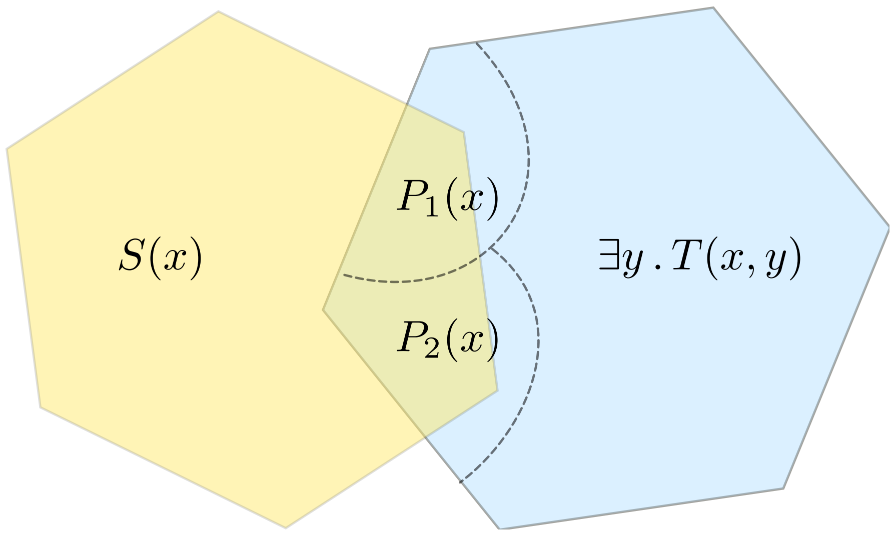


Figure 5.2: Region of validity computed for an example requiring AE-VAL to iterate two times.

to the form $S(x) \Rightarrow T(x,y)$ and then attempts to extend all models of $S(x)$ to models of $T(x,y)$. If such an extension is possible, then the input formula is valid, and a relationship between x and y are gathered in a Skolem function. Otherwise the formula is invalid, and no Skolem function exists.

Both of the synthesis algorithms presented in this thesis rely on the fact that during each run, AE-VAL iteratively creates a set of formulas $\{P_i(x)\}$, such that each $P_i(x)$ has a common model with $S(x)$ and $P_i(x) \Rightarrow \exists y\,.\,T(x,y)$. After n iterations, AE-VAL establishes a formula $R_n(x) \overset{\text{def}}{=} \bigvee_{i=1}^{n} P_i(x)$ which by construction implies $\exists y\,.\,T(x,y)$. If additionally $S(x) \Rightarrow R_n(x)$, the input formula is valid, and the algorithm terminates. Figure 5.2 shows a Venn diagram for an example of the opposite scenario: $R_2(x) = P_1(x) \vee P_2(x)$, but the input formula is invalid. However, models of each $S(x) \wedge P_i(x)$ can still be extended to a model of $T(x,y)$.

In general, if after n iterations $S(x) \wedge T(x,y) \wedge \neg R_n(x)$ is unsatisfiable, then AE-VAL terminates. Note that the formula $\forall x.\ S(x) \wedge R_n(x) \Rightarrow \exists y.\ T(x,y)$ is valid by construction at any iteration of the algorithm. We say that $R_n(x)$ is a *region of validity*,

Algorithm 5.1: JSYN-VG (A : assumptions, G : guarantees)

```
F(s) ← true                              ▷ Fixpoint of viable states;
while true do
    $\phi \leftarrow \forall s, i.\ (F(s) \wedge A(s,i) \Rightarrow \exists s'.G_T(s,i,s') \wedge F(s'))$;
    $\langle valid, validRegion, Skolem\rangle \leftarrow$ AE-VAL($\phi$);
    if valid then
        if $\exists s.G_I(s) \wedge F(s)$ then
            return ⟨REALIZABLE, Skolem, s, F⟩;
        else                             ▷ Empty set of initial or viable states
            return UNREALIZABLE;
    else                                 ▷ Extract region of validity $Q(s,i)$
        $Q(s,i) \leftarrow validRegion$;
        $\phi' \leftarrow \forall s.\ (F(s) \Rightarrow \exists i.A(s,i) \wedge \neg Q(s,i))$;
        $\langle \_, violatingRegion, \_\rangle \leftarrow$ AE-VAL($\phi'$);
        $W(s) \leftarrow violatingRegion$;
        $F(s) \leftarrow F(s) \wedge \neg W(s)$      ▷ Refine set of viable states;
end
```

and in this chapter, we are interested in the *maximal* regions of validity, i.e., the ones produced by disjoining all $\{P_i(x)\}$ produced by AE-VAL before termination and by conjoining it with $S(x)$. Throughout the paper, we assume that all regions of validity are maximal.

Lemma 6. *Let $R_n(x)$ be the region of validity returned by* AE-VAL *for formula $\forall s.\ S(x) \Rightarrow \exists y\,.\,T(x,y)$. Then $\forall x.\ S(x) \Rightarrow (R_n(x) \Leftrightarrow \exists y\,.\,T(x,y))$.*

Proof. ($\Rightarrow$) By construction of $R_n(x)$.

($\Leftarrow$) Suppose towards contradiction that the formula does not hold. Then there exists x_0 such that $S(x_0) \wedge (\exists y.T(x_0,y)) \wedge \neg R_n(x_0)$ holds. But this is a direct contradiction for the termination condition for AE-VAL. Therefore the original formula does hold. □

5.3 Validity-Guided Synthesis

Algorithm 5.1, named JSyn-vg (for *validity guided*), shows the validity-guided technique that we use towards the automatic synthesis of implementations. The specification is written using the Assume-Guarantee convention that we described in Chapter 3 and is provided as an input. For this algorithm we modify the form of the results provided by AE-VAL to $\langle x, y, z\rangle \leftarrow$ AE-VAL$(\ldots)$: x again specifies if the formula is (not) valid, y identifies the region of validity (in both cases), and z – the Skolem function (only in case of the validity).

The algorithm maintains a formula $F(s)$ which is initially assigned *true* (line 1). It then attempts to strengthen $F(s)$ until it only contains viable states (recall Definition 3.3.4 and Theorem 3.3.1), i.e., a greatest fixpoint is reached. We first encode Definition 3.3.4 in a formula ϕ and then provide it as input to AE-VAL (line 4) which determines its validity (line 5). If the formula is valid, then a witness *Skolem* is non-empty. By construction, it contains valid assignments to the existentially quantified variables of ϕ. In the context of viability, this witness is capable of providing viable states that can be used as safe reactions, given an input that satisfies the assumptions.

With the valid formula ϕ in hand, it remains to check that the fixpoint intersects with the initial states, i.e., to find a model of satisfiability of Theorem 3.3.1. If a model exists, it is directly combined with the extracted witness and used towards an implementation of the system, and the algorithm terminates (line 6). Otherwise, the contract is unrealizable since either there are no states that satisfy the initial state guarantees G_I, or the set of viable states F is empty.

If ϕ is not true for every possible assignment of the universally quantified variables, AE-VAL provides a *region of validity* $Q(s, i)$ (line 11). At this point, one might assume that $Q(s, i)$ is sufficient to restrict F towards a solution. This is not the case since $Q(s, i)$ creates a subregion involving both state and input variables. As such, it may contain constraints over the contract's inputs above what are required by A, ultimately leading to implementations that only work correctly for a small part of the input domain.

Fortunately, we can again use AE-VAL's capability of providing regions of validity towards removing inputs from Q. Essentially, we want to remove those states from Q if even one input causes them to violate the formula on line 3. We denote by W the *violating region* of Q. To construct W, AE-VAL determines the validity of formula $\phi' \leftarrow \forall s.\ (F(s) \Rightarrow \exists i.A(s,i) \wedge \neg Q(s,i))$ (line 12) and computes a new region of validity.

If ϕ' is invalid, it indicates that there are still non-violating states (i.e., outside W) that may lead to a fixpoint. Thus, the algorithm removes the unsafe states from $F(s)$ in line 8, and iterates until a greatest fixpoint for $F(s)$ is reached. If ϕ' is valid, then every state in $F(s)$ is unsafe, under a specific input that satisfies the contract assumptions (since $\neg Q(s,i)$ holds in this case), and the specification is unrealizable (i.e., in the next iteration, the algorithm will reach line 7).

5.3.1 Soundness

Lemma 7. $\mathsf{Viable} \Rightarrow F$ *is an invariant for Algorithm 5.1.*

Proof. It suffices to show this invariant holds each time F is assigned. On line 1, this is trivial. For line 8, we can assume that $\mathsf{Viable} \Rightarrow F$ holds prior to this line. Suppose towards contradiction that the assignment on line 8 violates the invariant. Then there exists s_0 such that $F(s_0)$, $W(s_0)$, and $\mathsf{Viable}(s_0)$ all hold. Since W is the region of validity for ϕ' on line 12, we have $W(s_0) \wedge F(s_0) \Rightarrow \exists i.A(s_0,i) \wedge \neg Q(s_0,i)$ by Lemma 6. Given that $W(s_0)$ and $F(s_0)$ hold, let i_0 be such that $A(s_0,i_0)$ and $\neg Q(s_0,i_0)$ hold. Since Q is the region of validity for ϕ on line 3, we have $F(s_0) \wedge A(s_0,i_0) \wedge \exists s'.G_T(s_0,i_0,s') \wedge F(s') \Rightarrow Q(s_0,i_0)$ by Lemma 6. Since $F(s_0)$, $A(s_0,i_0)$ and $\neg Q(s_0,i_0)$ hold, we conclude that $\exists s'.G_T(s_0,i_0,s') \wedge F(s') \Rightarrow \bot$. We know that $\mathsf{Viable} \Rightarrow F$ holds prior to line 8, thus $\exists s'.G_T(s_0,i_0,s') \wedge \mathsf{Viable}(s') \Rightarrow \bot$. But this is a contradiction since $\mathsf{Viable}(s_0)$ holds.

Therefore the invariant holds on line 8. □

Theorem 5.3.1. *The* REALIZABLE *and* UNREALIZABLE *results of Algorithm 5.1 are sound.*

Proof. If Algorithm 5.1 terminates, then the formula for ϕ on line 3 is valid. Rewritten, F satisfies the formula

$$\forall s.\ F(s) \Rightarrow \big(\forall i.\ A(s,i) \Rightarrow \exists s'.G_T(s,i,s') \wedge F(s')\big)\,. \tag{5.1}$$

Let the function f be defined over state predicates as

$$f = \lambda V.\lambda s.\ \forall i.\ A(s,i) \Rightarrow \exists s'.G_T(s,i,s') \wedge V(s'). \tag{5.2}$$

State predicates are equivalent to subsets of the state space and form a lattice in the natural way. Moreover, f is monotone on this lattice. From Equation 5.1 we have $\forall s.F(s) \Rightarrow f(F)(s)$. Thus F is a post-fixed point of f. In Definition 3.3.4, Viable is defined as the greatest fixed-point of f. Thus $F \Rightarrow \mathsf{Viable}$ by the Knaster-Tarski theorem. Combining this with Lemma 7, we have $F = \mathsf{Viable}$. Therefore the check on line 6 is equivalent to the check in Theorem 3.3.1 for realizability. □

5.3.2 Termination on finite models

Lemma 8. *Every loop iteration in Algorithm 5.1 either terminates or removes at least one state from F.*

Proof. It suffices to show that at least one state is removed from F on line 8. That is, we want to show that $F \cap W \neq \varnothing$ since this intersection is what is removed from F by line 8.

If the query on line 4 is valid, then the algorithm terminates. If not, then there exists a state s^* and input i^* such that $F(s^*)$ and $A(s^*, i^*)$ such that there is no state s' where both $G(s^*, i^*, s')$ and $F(s')$ hold. Thus, $\neg Q(s^*, i^*)$, and $s^* \in \mathit{violatingRegion}$, so $W \neq \varnothing$. Next, suppose towards contradiction that $F \cap W = \varnothing$ and $W \neq \varnothing$. Since W is the region of validity for ϕ' on line 12, we know that F lies completely outside the region of validity and therefore $\forall s.\ \neg\exists i. A(s,i) \wedge \neg Q(s,i)$ by Lemma 6. Rewritten, $\forall s, i.\ A(s,i) \Rightarrow Q(s,i)$. Note that Q is the region of validity for ϕ on line 3. Thus A is completely contained within the region of validity and formula ϕ is valid. This is a contradiction since if ϕ is valid then line 8 will not be executed in this iteration of the loop. Therefore $F \cap W \neq \varnothing$ and at least one state is removed from F on line 8. $\square$

Theorem 5.3.2. *For finite models, Algorithm 5.1 terminates.*

Proof. Immediately from Lemma 8 and the fact that AE-VAL terminates on finite models [37]. $\square$

5.3.3 Applying JSyn-vg to the Cinderella-Stepmother game

Figure 5.3 shows one possible interpretation of the contract designed for the instance of the Cinderella-Stepmother game that we introduced in Section 5.1. The contract is defined as a Lustre [61] node `game`, with a global constant `C` denoting the bucket capacity. The node describes the game itself, through the problem's input and output variables. The main input is Stepmother's distribution of one unit of water over five different input variables, `i1` to `i5`. While the node contains a sixth input argument, namely `e`, this is in fact used as the output of the system that we want to implement, representing Cinderella's choice at each of her turns.

We specify the system's inputs `i1`, ..., `i5` using the `REALIZABLE` statement and define the contract's assumptions over them: $A(i_1, \ldots, i_5) = (\bigwedge_{k=1}^{5} i_k >= 0.0) \wedge (\sum_{k=1}^{5} i_k =$

```
const C = 2.0;

-- empty buckets e and e+1 each round
node game(i1,i2,i3,i4,i5: real; e: int) returns (guarantee: bool);
var
  b1, b2, b3, b4, b5 : real;
let
  assert i1 >= 0.0 and i2 >= 0.0 and i3 >= 0.0 and i4 >= 0.0 and
        i5 >= 0.0;
  assert i1 + i2 + i3 + i4 + i5 = 1.0;

  b1 = 0.0 -> (if (e = 5 or e = 1) then i1 else (pre(b1) + i1));
  b2 = 0.0 -> (if (e = 1 or e = 2) then i2 else (pre(b2) + i2));
  b3 = 0.0 -> (if (e = 2 or e = 3) then i3 else (pre(b3) + i3));
  b4 = 0.0 -> (if (e = 3 or e = 4) then i4 else (pre(b4) + i4));
  b5 = 0.0 -> (if (e = 4 or e = 5) then i5 else (pre(b5) + i5));

  guarantee = b1 <= C and b2 <= C and b3 <= C and b4 <= C and
        b5 <= C;

  --%REALIZABLE i1, i2, i3, i4, i5;
  --%PROPERTY guarantee;
tel;
```

Figure 5.3: An Assume-Guarantee contract for the Cinderella-Stepmother game in Lustre.

1.0). The assignment to boolean variable `guarantee` (distinguished via the `PROPERTY` statement) imposes the guarantee constraints on the buckets' states through the entire duration of the game, using the local variables `b1` to `b5`. Initially, each bucket is empty, and with each transition to a new state, the contents depend on whether Cinderella chose the corresponding bucket or not. If so, the value of each `b`$_k$ at the the next turn becomes equal to the value of the corresponding input variable `i`$_k$. Formally, for the initial state, $G_I(C, b_1, \ldots, b_5) = (\bigwedge_{k=1}^{5} b_k = 0.0) \wedge (\bigwedge_{k=1}^{5} b_k \leq C)$, while the transitional guarantee is $G_T([C, b_1, \ldots, b_5, e], i_1, \ldots, i_5, [C, b'_1, \ldots, b'_5, e']) = (\bigwedge_{k=1}^{5} b'_k = ite(e = k \vee e = k_{prev}, i_k, b_k + i_k) \wedge (\bigwedge_{k=1}^{5} b'_k \leq C)$, where $k_{prev} = 5$ if $k = 1$, and $k_{prev} = k - 1$ otherwise. Interestingly, the lack of explicit constraints over e, i.e. Cinderella's choice, permits the action of Cinderella skipping her current turn, i.e. she does not choose to empty any of the buckets. With the addition of the guarantee $(e = 1) \vee \ldots \vee (e = 5)$, the contract is still realizable, and the implementation is verifiable, but Cinderella is not allowed to skip her turn anymore.

If the bucket was not covered by Cinderella's choice, then its contents are updated by Stepmother's addition, if any, to the volume of water that the bucket already had. The arrow (`->`) operator distinguishes the initial state (on the left) from subsequent states (on the right), and variable values in the previous state can be accessed using the `pre` operator. The contract should only be realizable if, assuming valid inputs given by the Stepmother (i.e. positive values to input variables that add up to one water unit), Cinderella can keep reacting indefinitely, by providing outputs that satisfy the guarantees (i.e. she empties buckets in order to prevent overflow in Stepmother's next turn). We provide the contract in Figure 5.3 as input to Algorithm 5.1 which then iteratively attempts to construct a fixpoint of viable states, closed under the transition relation.

Initially $F = \mathit{true}$, and we query AE-VAL for the validity of formula $\phi \overset{\text{def}}{=} \forall i_1, \ldots, i_5, b_1, \ldots, b_5 \,.\, A(i_1, \ldots, i_5) \Rightarrow \exists b'_1, \ldots, b'_5, e \,.\, G_T(i_1, \ldots, i_5, b_1, \ldots, b_5, b'_1, \ldots, b'_5, e)$. Since F is empty, there are states satisfying A, for which there is no transition to G_T. In particular, one such counterexample identified by AE-VAL is represented by the set of assignments $\mathit{cex} = \{\ldots, b_4 = 3025, i_4 = 0.2, b'_4 = 3025.2, \ldots\}$, where the already overflown bucket b_4 receives additional water during the transition to the next state, violating the contract guarantees. In addition, AE-VAL provides us with a region of validity $Q(i_1, \ldots, i_5, b_1, \ldots, b_5)$, a formula for which $\forall i_1, \ldots, i_5, b_1, \ldots, b_5 \,.\, A(i_1, \ldots, i_5) \wedge Q(i_1, \ldots, i_5, b_1, \ldots, b_5) \Rightarrow \exists b'_1, \ldots, b'_5, e \,.\, G_T(i_1, \ldots, i_5, b_1, \ldots, b_5, b'_1, \ldots, b'_5, e)$ is valid. Precise encoding of Q is too large to be presented; intuitively it contains some constraints on $i_1, \ldots, i_5$ and $b_1, \ldots, b_k$ which are stronger than A and which block the inclusion of violating states such as the one described by cex.

Since Q is defined over both state and input variables, it might contain constraints over the inputs, which is an undesirable side-effect. In the next step, AE-VAL decides the validity of formula $\forall b_1, \ldots, b_5 \,.\, \exists i_1, \ldots, i_5 \,.\, A(i_1, \ldots, i_5) \land \neg Q(i_1, \ldots, i_5, b_1, \ldots, b_5)$ and extracts a violating region W over $b_1, \ldots, b_5$. Precise encoding of W is also too large to be presented; intuitively it captures certain steps in which Cinderella may not take the optimal action. Blocking them leads us eventually to proving the contract's realizability.

From this point on, the algorithm continues following the steps explained above. In particular, it terminates after one more refinement, at depth 2. At that point, the refined version of ϕ is valid, and AE-VAL constructs a witness containing valid reactions to environment behavior. In general, the witness is described through the use of nested *if-then-else* blocks, where the conditions are subsets of the antecedent of the implication in formula ϕ, while the body contains valid assignments to state variables to the corresponding subset.

5.4 Implementation and Evaluation

The implementation of the algorithm has been added to a branch of the JKind [47] model checker[1]. JKind officially supports synthesis using a k-inductive approach, named JSyn [62]. For clarity, we named our validity-guided technique JSyn-vg (i.e., validity-guided synthesis). JKind uses Lustre [61] as its specification and implementation language. JSyn-vg encodes Lustre specifications in the language of linear real and integer arithmetic (LIRA) and communicates them to AE-VAL[2]. Skolem functions returned by AE-VAL then get translated into an efficient and practical implementation. To compare the quality of implementations with those produced by JSyn, we use SMTLib2C,

[1]The JKind fork with JSyn-vg is available at https://goo.gl/WxupTe.
[2]The AE-VAL tool is available at https://goo.gl/CbNMVN.

a tool that has been specifically developed to translate Skolem functions to C implementations[3].

5.4.1 Experimental results

We evaluated JSyn-vg by synthesizing implementations for 124 contracts[4] originating from a broad variety of contexts. Since we have been unable to find past work that contained benchmarks directly relevant to our approach, we propose a comprehensive collection of contracts that can be used by the research community for future advancements in reactive system synthesis for contracts that rely on infinite theories. Our benchmarks are split into three categories:

- 59 contracts correspond to various industrial projects, such as a Quad-Redundant Flight Control System, a Generic Patient Controlled Analgesia infusion pump, as well as a collection of contracts for a Microwave model, written by graduate students as part of a software engineering class;
- 54 contracts were initially used for the verification of existing handwritten implementations [55];
- 11 contracts contain variations of the Cinderella-Stepmother game, as well as examples that we created.

All of the synthesized implementations were verified against the original contracts using JKind.

The goal of this experiment was to determine the performance and generality of the JSyn-vg algorithm. We compared against the existing JSyn algorithm, and for the Cinderella model, we compared against [6] (this was the only reactive synthesis problem in that paper). We examined the following aspects:

[3]The SMTLib2C tool is available at https://goo.gl/EvNrAU.
[4]All of the benchmark contracts can be found at https://goo.gl/2p4sT9.

	JSyn	JSyn-vg
Problems solved	113	**124**
Synthesis time (avg - seconds)	5.72	**2.78**
Synthesis time (max - seconds)	352.1	**167.55**
Implementation Size (avg - Lines of Code)	72.88	**70.66**
Implementation Size (max - Lines of Code)	2322	**2142**
Implementation Performance (avg - ms)	57.84	**56.32**
Implementation Performance (max - ms)	485.88	**459.95**

Table 5.1: Benchmark statistics.

- time required to synthesize an implementation;
- size of generated implementations in lines of code (LoC);
- execution speed of generated C implementations derived from the synthesis procedure; and
- number of contracts that could be synthesized by each approach.

Since JKind already supports synthesis through JSyn, we were able to directly compare JSyn-vg against JSyn's k-inductive approach. We ran the experiments using a computer with Intel Core i3-4010U 1.70GHz CPU and 16GB RAM.

A listing of the statistics that we tracked while running experiments is presented in Table 5.1. Figure 5.4a shows the time taken by JSyn and JSyn-vg to solve each problem, with JSyn-vg outperforming JSyn for the vast majority of the benchmark suite, often times by a margin greater than 50%. Figure 5.4b on the other hand, depicts small differences in the overall size between the synthesized implementations. While it would be reasonable to conclude that there are no noticeable improvements, the big picture is different: solutions by JSyn-vg always require just a single Skolem function, but solutions by JSyn may require several ($k-1$ to initialize the system, and one for the inductive step). In our evaluation, JSyn proved the realizability of the majority of benchmarks by constructing proofs of length $k = 0$, which essentially means that the

Game	JSyn-vg			ConSynth [6]	
	Impl. Size (LoC)	Impl. Performance (ms)	Time	Time (Z3)	Time (Barcelogic)
Cind (C = 3)	204	128.09	4.5s	3.2s	1.2s
Cind2 (C = 3)	2081	160.87	28.7s		
Cind (C = 2)	202	133.04	4.7s	1m52s	1m52s
Cind2 (C = 2)	1873	182.19	27.2s		

Table 5.2: Cinderella-Stepmother results.

entire space of states is an inductive invariant. However, several spikes in Figure 5.4b refer to benchmarks, for which JSyn constructed a proof of length $k > 0$, which was significantly longer that the corresponding proof by JSyn-vg. Interestingly, we also noticed cases where JSyn implementations are (insignificantly) shorter. This provides us with another observation regarding the formulation of the problem for $k = 0$ proofs. In these cases, JSyn proves the existence of viable states, starting from a set of *pre-initial* states, where the contract does not need to hold. This has direct implications to the way that the $\forall\exists$-formulas are constructed in JSyn's underlying machinery, where the assumptions are "baked" into the transition relation, affecting thus the performance of AE-VAL.

One last statistic that we tracked was the performance of the synthesized C implementations in terms of single step of execution of the system, which can be seen in Figure 5.4c. The performance was computed as the mean of 1000000 iterations of executing each implementation using random input values. According to the figure as well as Table 5.1, the differences are minuscule on average.

Figure 5.4 does not cover the entirety of the benchmark suite. From the original 124 problems, eleven of them cannot be solved by JSyn's k-inductive approach. Four of these files are variations of the Cinderella-Stepmother game using different representations of the game, as well as two different values for the bucket capacity (2 and 3). Using the variation in Figure 5.3 as an input to JSyn, we receive an "unrealizable" answer, with the counterexample shown in Figure 5.5. Reading through the feedback provided by JSyn, it is apparent that the underlying SMT solver is incapable of choosing the correct buckets to empty, leading eventually to a state where an overflow occurs for the third bucket. As we already discussed though, a winning strategy exists for the Cinderella game, as long as the bucket capacity C is between 1.5 and 3. This provides an excellent demonstration of the inherent weakness of JSyn for determining unrealizability. JSyn-vg's validity-guided approach, is able to prove the realizability for these contracts, as well as synthesize an implementation for each.

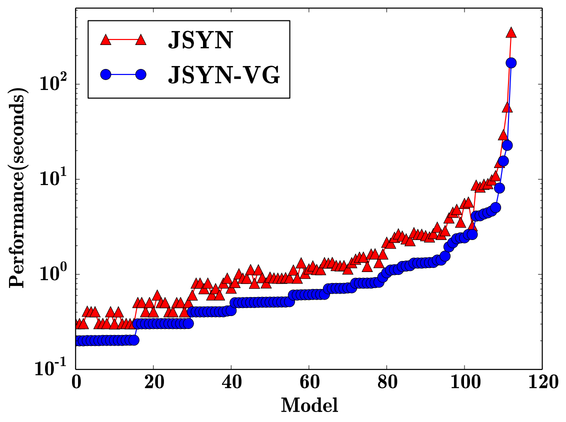
Table 5.2 shows how JSyn-vg performed on the four contracts describing the Cinderella-Stepmother game. We used two interpretations for the game, differing in terms of the number of variables and auxillary functions that are used, and exercised both for the cases where the bucket capacity C is equal to 2 and 3. Regarding the synthesized implementations, their size is proportional to the complexity of the program (Cind2 contains more local variables and a helper function to empty buckets). Despite this, the implementation performance remains the same across all implementations. Finally for reference, the table contains the results from the template-based approach followed in Consynth [6]. From the results, it is apparent that providing templates yields better performance for the case of $C = 3$, but our approach overperforms Consynth when it comes to solving the harder case of $C = 2$. Finally, the original paper for Consynth also explores the synthesis of winning strategies for Stepmother using the liveness property that a bucket will eventually overflow. While JKind does not natively support liveness properties, we successfully synthesized an implementation for Stepmother using a bounded notion of liveness with counters. We leave an evaluation of this category of specifications for future work.

Overall, JSyn-vg's validity-guided approach provides significant advantages over the k-inductive technique followed in JSyn, and effectively expands JKind's solving capabilities regarding specification realizability. On top of that, it provides an efficient "hands-off" approach that is capable of solving complex games. The most significant contribution however, is the applicability of this approach, as it is agnostic to the exercised theory and can be extended in the future to support other kinds of specifications.

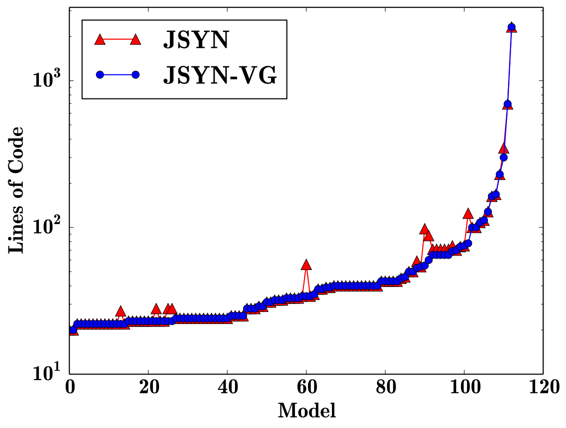
5.5 Conclusion

This chapter presented a novel and elegant approach towards the synthesis of reactive systems, using only the knowledge provided by the system specification expressed in infinite theories. The main goal is to converge to a fixpoint by iteratively blocking subsets of unsafe states from the problem space. This is achieved through the continuous extraction of regions of validity which hint towards subsets of states that lead to a candidate implementation.

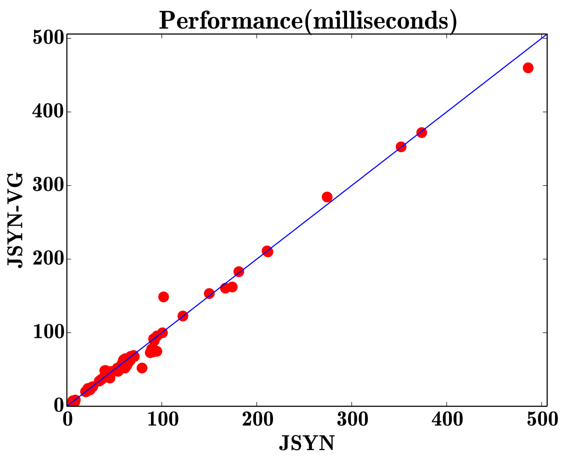
This is the first complete attempt, to the best of our knowledge, on handling valid subsets of a $\forall\exists$-formula to construct a greatest fixpoint on specifications expressed using infinite theories. We were able to prove its effectiveness in practice, by comparing it to an already existing approach that focuses on constructing k-inductive proofs of realizability, as well as CONSYNTH, a state-of-the-art synthesizer for infinite games. We showed how the new algorithm performs better than the k-inductive approach, both in terms of performance as well as the soundness of results.



(a) Performance of synthesizers



(b) Size of implementations



(c) Performance of implementations

Figure 5.4: Experimental results.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     UNREALIZABLE || K = 6 || Time = 2.017s
                Step
     variable      0    1      2      3      4      5
     INPUTS
     i1            0    0      0 0.416* 0.944* 0.666*
     i2            1    0 0.083* 0.083*      0 0.055*
     i3            0    1 0.305*    0.5 0.027* 0.194*
     i4            0    0 0.611*      0      0 0.027*
     i5            0    0      0      0 0.027* 0.055*

     OUTPUTS
     e             1    3      1      5      4      5

     NODE OUTPUTS
     guarantee   true true   true   true   true  false

     NODE LOCALS
     b1            0    0      0 0.416* 1.361* 0.666*
     b2            0    0 0.083* 0.166* 0.166* 0.222*
     b3            0    1 1.305* 1.805* 1.833* 2.027*
     b4            0    0 0.611* 0.611*      0 0.027*
     b5            0    0      0      0 0.027* 0.055*

     * display value has been truncated
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 5.5: Spurious counterexample for Cinderella-Stepmother example using JSyn

Chapter 6

Synthesis of Infinite-State Reactive Systems with Random Behavior

6.1 Introduction

In the previous two chapters we discussed the problem of program synthesis from formal specifications. Synthesized implementations are generated from a proof of the specification's realizability and typically have the form of *deterministic witnesses*. Thus, by design they always compute (1) an output that meets the specification, and (2) the same output for each particular input. Determinism, however, prevents us from synthesizing systems that take advantage of *randomness* to diversify their behavior[1]. Advantages offered by these systems can be better understood when put into context of their applications. For the purposes of this dissertation we consider two such promising areas, robot motion planning and fuzz testing.

Fuzz testing. Synthesis of random designs allows one to specify and create system-specific *fuzzers* [73]. The idea is to follow a mindset similar to how *model-based testing*

[1]For the sake of brevity throughout the chapter, we refer to such systems using the adjective *random* (e.g. *random* system/design/witness/controller).

techniques utilize the system-under-test (SUT) specification to generate test cases [100]. We propose to use the fragment of the model related to the SUT inputs to synthesize a fuzzer that repeatedly generates random (and sometimes malformed) tests. This fragment can be alternatively viewed as the fuzzer's specification, which can be further enriched with properties that dictate its behavior when certain testing objectives are met. For example, when a vulnerability is detected, we can limit the fuzzer's next generated test cases within a desired range around the test that exposed the issue. System coverage is also one such objective, where we can dictate how the fuzzer diversifies the generated tests through its specification, improving the chances of reaching previously unexplored system states. From a qualitative standpoint, synthesis in model-based fuzz testing can be considered as a viable high-level solution that does not require the user to create extensive corpora of tests. Furthermore, the synthesized fuzzers can be strong, SUT-specific alternatives to general-purpose model-based fuzzers. [3, 58, 101].

Robot motion planning. In coverage path planning problems, the goal is to maximize the area that a robot can cover while avoiding obstacles [50]. Furthermore, randomness can serve as an additional security barrier in avoidance games that involve adversaries with learning capabilities. A random strategy is typically harder to infer and exploit. In the special case of infinite-state problems, it is an even bigger challenge, as the current state-of-the-art in automata learning is limited to finite-state problems [60].

We treat systems in the aforementioned applications as so-called *reactive systems*, which have to exhibit specification-compliant behavior against an unpredictable environment. Examples are commonly found in aviation, autonomous vehicles, and medical devices. Synthesis of random reactive designs is offered by the recent Reactive Control Improvisation (RCI) [44, 46] framework, but limited only to finite-state systems (i.e., over the boolean domain), relies on probabilistic analysis to determine the realizability of the specification, and its synthesized witnesses require further refinement to be applicable in real world scenarios.

We present a novel approach to synthesis of random infinite-state systems, whose corresponding specifications may involve constraints over the Linear Integer or Real Arithmetic theories (LIRA) [5] and thus not limited to finite-state systems. The intuition behind this effort is to allow reasoning, and consequently synthesis, over ranges of safe reactions instead of computing witnesses with deterministic responses.

The pursuit of generality poses new challenges, for which we propose a novel Skolemization procedure to simulate randomness. We built this procedure on top of JSyn-vg [63], the validity-guided synthesis algorithm presented in Chapter 5. It iteratively generates a greatest fixpoint over system states that ensures the realizability of the given specification but offers only a brute and inflexible strategy for witness extraction (via predetermined Skolemization rules) [35, 36, 38]. Our key novelty is in a new algorithm that enables replacing deterministic assignments in the Skolem functions with applications of *uninterpreted random number generators*. Uninterpreted functions allow us to reason about solutions with random, broad, and most importantly, compliant behavior.

The new Skolem extraction algorithm preserves JSyn-vg's important properties. Thus, the procedure remains completely automated, unlike previous work on infinite-state synthesis that requires additional templates, or the user's intervention [2, 7, 39, 91]. More importantly, our work imposes no performance overheads over JSyn-vg, remaining thus competitive with other state-of-the-art tools which could be considered for random synthesis [79]. We implemented the Skolem extraction algorithm and applied it in two distinct case studies.

Model-based fuzz testing. We are the first to explore the applicability of reactive synthesis in fuzz testing. On a chosen set of applications designed for the DARPA Cyber Grand Challenge [43, 72], the synthesized fuzzers performed competitively against well-established tools (AFL [103], AFLFast [10]), both in terms of code coverage as well as exposing vulnerabilities.

Robot motion planning. We synthesized safe robot controllers that participate in avoidance games on both bounded and infinite arenas. Using simulation, we show how the synthesized controller leads to the robot being capable of avoiding its adversary while moving in random patterns. We demonstrate how the synthesized strategies are safe by design, no matter what bias is introduced at the implementation level. Furthermore, we showcase why randomness in the controller behavior is a mandatory feature, if synthesis is to be considered for coverage path planning problems.

To summarize, the contributions of this work are:

- the first complete formal framework that enables specification and synthesis of random infinite-state reactive systems;
- a novel Skolemization procedure that enables random synthesis with no performance overhead, by taking advantage of uninterpreted functions to reason about ranges of valid reactions;
- a novel application of synthesis in model-based fuzz testing, where we generated reactive fuzzers, yielding competitive results in terms of system coverage and vulnerability detection; and
- the application of synthesized random controllers in safety problems for robot motion planning, outlining important advantages over deterministic solutions.

The rest of the paper is structured as follows. Section 6.2 provides the necessary formal background on which our work depends. Section 6.3 illustrates and Section 6.4 describes in detail the algorithm for synthesis of random Skolem functions. The implementation is outlined in Section 6.5 and the case studies are presented in Sect 6.6

and Section 6.7, with an evaluation of synthesis time in Section 6.8. Finally, we discuss related work in Section 6.9 and conclude in Section 6.10.

6.2 Background and Notation

A first-order formula φ is satisfiable if there exists an assignment m, called a model, under which φ evaluates to $\top$ (denoted $m \models \varphi$). If every model of φ is also a model of ψ, then we write $\varphi \Rightarrow \psi$. A formula φ is called *valid* if $\top \Rightarrow \varphi$. For existentially-quantified formulas of the form $\exists y \,.\, \psi(x, y)$, validity requires that each assignment of variables in x can be *extended* to a model of $\psi(x, y)$. For a valid formula $\exists y \,.\, \psi(x, y)$, a term $sk_y(x)$ is called a *Skolem*, if $\psi(x, sk_y(x))$ is valid. More generally, for a valid formula $\exists \vec{y} \,.\, \psi(x, \vec{y})$ over a vector of existentially quantified variables $\vec{y}$, there exists a vector of individual Skolem terms, one for each variable $\vec{y}[j]$, where $0 < j \leq N$ and $N = |\vec{y}|$, such that: $\top \Rightarrow \psi(x, sk_{\vec{y}[1]}(x), \ldots, sk_{\vec{y}[N]}(x))$.

6.2.1 Synthesis with JSyn-vg

The work presented in this paper is based on JSyn-vg, the reactive synthesis procedure for specifications in the form of *Assume-Guarantee contracts* that was presented in Chapter 5. To keep the present chapter self-contained we summarize the algorithm and its main characteristics in this section.

Systems are described in terms of inputs $\vec{x}$ and outputs $\vec{y}$, using the predicate $I(\vec{y})$ to denote the set of initial outputs and $T(\vec{y}, \vec{x}, \vec{y}')$ for the system's transition relation, where the next (primed) outputs $\vec{y}'$ depend on the current input and state. *Assumptions* $A(\vec{x}, \vec{y})$ correspond to assertions over the system's current state, while the set of *guarantees* is decomposed into constraints over the initial outputs $G_I(\vec{y})$, and guarantees $G_T(\vec{y}, \vec{x}, \vec{y}')$ that have to hold over any valid transition (i.e., with respect to $T(\vec{y}, \vec{x}, \vec{y}')$).

Algorithm 6.1: JSYN-VG $\Big(A(\vec{x},\vec{y}), G_I(\vec{y}), G_T(\vec{y},\vec{x},\vec{y}')\Big)$, cf. [63].

```
Input: $A(\vec{x},\vec{y})$: assumptions, $G_I(\vec{y}), G_T(\vec{y},\vec{x},\vec{y}')$: guarantees
Output: $\langle$REALIZABLE, $Skolem\rangle$/UNREALIZABLE
1 $F(\vec{y}) \leftarrow \top$;
2 while $\top$ do
3     $\phi \leftarrow \forall\vec{x},\vec{y}.\ (F(\vec{y}) \wedge A(\vec{x},\vec{y}) \Rightarrow \exists\vec{y}'.G_T(\vec{y},\vec{x},\vec{y}') \wedge F(\vec{y}'))$;
4     $\langle valid, validRegion(\vec{x},\vec{y}), Skolem\rangle \leftarrow$ AE-VAL($\phi$);
5     if $valid$ then
6         if $\exists\vec{y}.G_I(\vec{y}) \wedge F(\vec{y})$ then return $\langle$REALIZABLE, $Skolem\rangle$;
7         else return UNREALIZABLE;
8     else $F(\vec{y}) \leftarrow F(\vec{y}) \wedge \neg$EXTRACTUNSAFE($validRegion(\vec{x},\vec{y})$);
9 end
```

The algorithm behind JSYN-VG performs a realizability analysis to determine the existence of a greatest fixpoint of states meeting the contract, that can lead to an implementation. Furthermore, the computed fixpoint can be directly used for the purposes of synthesis, as it precisely captures a collection of system output constraints which, when instantiated, define safe reactions. Formally, the computed fixpoint is a set of *viable* outputs, guaranteed to preserve safety by requiring that a valid transition to another viable output is always available.

$$\mathsf{Viable}(\vec{y}) \overset{\text{def}}{=} \forall\vec{x},\vec{y}.(A(\vec{x},\vec{y}) \Rightarrow \exists\vec{y}'.\ G_T(\vec{y},\vec{x},\vec{y}') \wedge \mathsf{Viable}(\vec{y}')) \tag{6.1}$$

The coinductive definition of viable states is sufficient to prove the realizability of a contract, as long as the corresponding decision procedure can find a viable output that satisfies the initial guarantees $G_I(\vec{y})$:

$$\exists\vec{y}.G_I(\vec{y}) \wedge \mathsf{Viable}(\vec{y})$$

Given a proof of the contract's realizability, the problem of synthesis is formally defined as the process of computing an initial output $\vec{y}_{init}$ and a function $f(\vec{x},\vec{y})$ such that $G_I(\vec{y}_{init})$ and $\forall \vec{x}, \vec{y}. \mathsf{Viable}(\vec{y}) \Rightarrow \mathsf{Viable}(f(\vec{x}, \vec{y}))$ hold true.

Algorithm 6.1 summarizes JSYN-VG. It begins with the generic candidate fixpoint $F(\vec{y}) = \top$, and solves the $\forall\exists$-formula ϕ for the validity (line 4) that corresponds to the definition of viable outputs in Eq. 6.1. If ϕ is valid and an output vector in $F(\vec{y})$ exists that satisfies the initial guarantees, then the contract is declared realizable, and a witnessing Skolem term is extracted. If ϕ is invalid, the algorithm extracts a maximal set of states as $\mathit{validRegion}(\vec{x}, \vec{y}) \subset F(\vec{y}) \wedge A(\vec{x}, \vec{y})$, for which ϕ is valid. Due to the possibility of $\mathit{validRegion}(\vec{x}, \vec{y})$ strengthening the assumptions $A(\vec{x}, \vec{y})$, an additional step on $\mathit{validRegion}(\vec{x}, \vec{y})$ is used to extract a set of constraints over unsafe states (EXTRACTUNSAFE). The negation of this set is then added as a new conjunct to the candidate $F(\vec{y})$ and the algorithm iterates until either ϕ is valid or $F(\vec{y}) = \bot$. For further details, we refer the reader to Chapter 5 on JSYN-VG [63].

6.2.2 Realizability and Synthesis with AE-VAL

The greatest fixpoint algorithm described by JSYN-VG uses AE-VAL, a specialized decision procedure to determine the validity of $\forall\exists$-formulas, as well as generate deterministic witnesses in the form of Skolem terms. The latter feature is also the point of interest behind this work, as randomness is supported naturally through our proposed Skolem extraction algorithm.

Algorithm 6.2 gives a brief pseudocode of AE-VAL[2]. The procedure lazily derives a sequence of Model-Based Projections (MBPs) [8] that decompose the problem, where each model gives rise to a *precondition* that captures an arbitrary subspace on the universally-quantified variables and a corresponding collection of *Skolem constraints* for the existentially-quantified variables (line 6, vector formulas *pre* and π, respectively).

[2]Note that for simplicity of presentation, in the pseudocode we assume a single existentially quantified variable y (however, the algorithm and the implementation can handle any vector $\vec{y}$).

Skolem constraints have the form of a conjunction (which by construction can be composed only of arithmetic relations) and serve as the building blocks towards generating the actual Skolem term. The ExtractSk procedure, used for Skolem extraction, implements an inflexible strategy to turn conjunctive Skolem constraints to Skolem terms. The final Skolem term has a form of a *decision tree*, where preconditions are placed on the nodes and local Skolem terms (i.e., outputs of ExtractSk) are on the leaves, i.e., the nested if-then-else structure ($ite(\cdot)$):

$$\begin{aligned} sk_y(\vec{x}) \overset{\text{def}}{=} ite(&pre[1], sk_{1,y}(\vec{x}), ite(pre[2], sk_{2,y}(\vec{x}), \ldots, \\ &(ite(pre[M-1], sk_{M-1,y}(\vec{x}), sk_{M,y}(\vec{x}))))) \end{aligned}$$

Finally, in the case that the input formula $\forall \vec{x} \exists y \,.\, \psi(\vec{x}, y)$ is invalid, AE-VAL returns $\bigvee_{i=1}^{M-1} pre[i](\vec{x})$ as the formula's maximal *region of validity*, i.e., the maximal subset of the universally quantified variables for which the formula becomes valid. This region is used by JSyn-vg in order to further refine the candidate fixpoint during each of its iterations (Algorithm 6.1, line 8).

6.3 Random Synthesis - Motivating Example

In this section, we demonstrate a complete run of the synthesis procedure and show how the standard synthesized witnesses are unable to exhibit random behavior. As an example, we use a safety robot motion planning problem from Neider et al. [79]. In this problem, a robot is placed on a one-dimensional grid with two players, the environment and the system, controlling its movement. Each player can choose to either move the robot left or right, or not move it at all (we refer to these choices using the values $-1, 0, 1$). The robot starts at $position = 0$, and the safety property for the system is to retain the robot in the area of the grid for which $position \geq 0$.

Algorithm 6.2: AE-VAL$\Big(\forall \vec{x} \exists y \,.\, \psi(\vec{x}, y)\Big)$, cf. [36, 38].

```
Input: $\forall \vec{x} \exists y \,.\, \psi(\vec{x}, y)$
Data: MBPs $pre$, Skolem constraints $\pi$
Output: Return value $\in \{valid, invalid\}$ of $\forall \vec{x} \exists y \,.\, \psi(\vec{x}, y)$, $validRegion$, $Skolem$
1 $M \leftarrow 1$;
2 while $\top$ do
3     if $\bigwedge_{i=1}^{M-1} \neg pre[i](\vec{x}) \Rightarrow \bot$ then
4         return $\langle valid, \top, \textsc{DecisionTree}(pre, \textsc{ExtractSk}(\vec{x}, y, \pi))\rangle$;
5     if $\exists m \models \psi(\vec{x}, y) \wedge \bigwedge_{i=1}^{M-1} \neg pre[i](\vec{x})$ then
6         $pre[M](\vec{x}), \pi[M](\vec{x}, y) \leftarrow \textsc{GetMBP}(\vec{x}, y, m, \psi)$;
7         $M \leftarrow M + 1$;
8     else return $\langle invalid, \bigvee_{i=1}^{M-1} pre[i](\vec{x}), \varnothing\rangle$;
```

Figure 6.1 shows an Assume-Guarantee contract for the example, described in the Lustre language. The contract has the singleton input $\vec{x} = \{x\}$ (internally identified by the `--%REALIZABLE` statement) and the outputs $\vec{y} = \{y, position\}$[3]. The contract assumption is that the environment will only make legal choices, i.e., $A(\vec{x}, \vec{y}) \overset{\text{def}}{=} -1 \leq x \wedge x \leq 1$. The initial guarantee refers to the initial position of the robot and the system choices for movement, i.e., $G_I(\vec{y}) \overset{\text{def}}{=} (position = 0) \wedge (-1 \leq y \wedge y \leq 1)$. On the other hand, the transitional guarantee captures the safety property along with the stateful computation step for the new position, i.e., $G_T(\vec{y}, \vec{x}, \vec{y}') \overset{\text{def}}{=} (position' = position + x + y') \wedge (position' \geq 0)$ (the transition relation for *position* is defined using Lustre's `->` and `pre` operators).[4] The safety properties are captured by `ok1` and `ok2` (declared as such using `--%PROPERTY`).

The procedure begins with a call to JSYN-VG using the contract as its input. The contract is realizable and the greatest fixpoint of viable states is $F(\vec{y}) \overset{\text{def}}{=} property \geq 0$.

[3]Variable *property* is local to the contract. Formally, local variables are treated as system outputs.

[4]Intuitively, the problem is formalized in a manner where the position of the robot is updated after both players make a choice, with the system reacting to the choice of the environment.

```
node onedim(x,y : int) returns();
var
  ok1, ok2 : bool;
  position : int;
let
  assert x >= -1 and x <= 1;
  position =  0 -> (pre(position) + x + y);
  ok1 = y >= -1 and y <= 1;
  ok2 = position >= 0;

  --%PROPERTY ok1;
  --%PROPERTY ok2;
  --%REALIZABLE x;
tel;
```

Figure 6.1: Assume-Guarantee contract in Lustre.

```
void skolem() {
  if(position + x == 1) {
    y = -1;
  } else if (position + x >= -1 &&
        position + x <= 0) {
    y =- (position + x);
  } else {
    y = 0;
  }
}
```

Figure 6.2: Synthesized deterministic witness in C.

AE-VAL declares that the formula $\phi \stackrel{\text{def}}{=} \forall\vec{y},\vec{x}.(A(\vec{x},\vec{y}) \wedge F(\vec{y}) \Rightarrow \exists\vec{y}'.G_T(\vec{y},\vec{x},\vec{y}') \wedge F(\vec{y}')$ is valid and extracts a Skolem term as a witness. Figure 6.2 presents a direct translation of the function to C. The synthesized implementation behaves in a deterministic way under the following conditions:

1. whenever $\mathit{position} + x = 1$, the system chooses to move left ($y = -1$);
2. if $\mathit{position} + x$ equals 0 or -1, then the system chooses to do nothing or move right, respectively ($y = -(\mathit{position} + x)$);
3. for any other case, the system chooses to do nothing ($y = 0$).

While the implementation preserves safety, the set of possible actions are limited due to the deterministic assignments to the output y. Interestingly, for this particular implementation the system forces the robot to go back to positions that are dangerously close to the unsafe region! Similarly, the corresponding solution by Neider et al. is the set of positions in $[0, 3)$ (the authors refer to it as the *winning set*), which would translate to implementations where the system would never move the robot beyond $position = 2$. Nevertheless, implementations exist for which the system can exercise a broader set of behaviors. For this example in particular, when either condition (1) or (3) is true, the system can freely choose any possible move action without violating the safety properties. Figure 6.3 shows an implementation that can (theoretically) exercise any such possible assignment (we explain why in Section 6.4.2). In the following sections, we present a new method to synthesize a random witness that can, in theory, provide all such possible permutations using a single implementation.

Algorithm 6.3: ExtractSk($\vec{x}, y, \pi$)

```
Input: Variables $\vec{x}, y$, Skolem constraints $\pi(\vec{x}, y) = \bigwedge_{r \in E \cup D \cup G \cup GE \cup L \cup LE} r(\vec{x}, y)$
Output: Term $sk$, such that $(y = sk(\vec{x})) \Rightarrow \pi(\vec{x}, y)$
$\ell_{closed} \leftarrow \bot, u_{closed} \leftarrow \bot$;
if $E \neq \varnothing$ then return ASN($e$), s.t. $e \in E$;
if $G \cup GE \neq \varnothing$ then
    $\ell \leftarrow$ MAX($G \cup GE$);
    $\ell_{closed} \leftarrow G = \varnothing \vee$ MAX($G$) $<$ MAX($GE$);
if $L \cup LE \neq \varnothing$ then
    $u \leftarrow$ MIN($L \cup LE$);
    $u_{closed} \leftarrow L = \varnothing \vee$ MIN($L$) $>$ MIN($LE$);
if $\ell(\vec{x}) = u(\vec{x})$ then return $\ell$;
$H \leftarrow \langle$ASN($d$) $\mid d \in D\rangle$;
if $\ell =$ undef $\wedge$ $u =$ undef then return $f_{\text{RNG}}(H, \top, \top, -\infty, +\infty)$;
if $\ell =$ undef then return $f_{\text{RNG}}(H, \top, u_{closed}, -\infty, u)$;
if $u =$ undef then return $f_{\text{RNG}}(H, \ell_{closed}, \top, \ell, +\infty)$;
return $f_{\text{RNG}}(H, \ell_{closed}, u_{closed}, \ell, u)$
```

6.4 Synthesizing random designs

The standard Skolem term extraction algorithm in AE-VAL does not support the generation of Skolem functions with random variable assignments. In this section, we present a new procedure to compute witnesses that can be used to simulate random behavior.

6.4.1 Overview

Our proposed algorithm preserves the overall structure of AE-VAL as well as the soundness of its results [36]. The main idea is to replace the deterministic assignments that eventually appear in the leaves of the generated decision tree with applications of uninterpreted functions, which when translated at the implementation level, can be viewed as function calls to a user-defined random number generator. We refer to these functions as *uninterpreted random number generators*:

Definition 6.4.1 (Uninterpreted Random Number Generator)**.** An uninterpreted random number generator (URNG) is an uninterpreted function

$$f_{\text{RNG}}(H, \ell_{closed}, u_{closed}, l, u) : T_1 \times \ldots \times T_{|D|} \times \mathbb{B} \times \mathbb{B} \times T \times T \to T,$$

where $T, T_i : \{\mathbb{Z}, \mathbb{R}\}$, H is a collection of right side expressions extracted from the set of disequalities D, ℓ and u determine the bounded interval for the randomly generated value, and $\ell_{closed}, u_{closed}$ are boolean flags that, when set, identify the corresponding bound as being closed. Furthermore, we require the following postconditions to hold, on any supplied implementation of f_{RNG}:

1. $\forall h \in H . f_{\text{RNG}}(H, \_, \_, \_, \_) \neq h$
2. $f_{\text{RNG}}(H, \bot, \bot, \ell, u) \in (\ell, u)$
3. $f_{\text{RNG}}(H, \bot, \top, \ell, u) \in (\ell, u]$

4. $f_{\text{RNG}}(H, \top, \bot, \ell, u) \in [\ell, u)$

5. $f_{\text{RNG}}(H, \top, \top, \ell, u) \in [\ell, u]$

The use of URNGs allows us to reason about valid regions of values for variable assignments instead of a particular value. Furthermore, the postconditions defined for these functions play an integral role in determining the soundness of the resulting Skolem function. It is important to note that we do not have to reason regarding the emptiness of the intervals. The intuition behind this is that such computed constraints infer an unrealizable contract. In these scenarios AE-VAL would declare the input $\forall\exists$-formula as invalid, and the Skolem extraction algorithm would never be invoked.

6.4.2 Algorithm

Algorithm 6.3 shows our proposed procedure for extracting Skolem functions that allow for random choices. It is invoked from Algorithm 6.2 and takes a set of universally quantified variables $\vec{x}$, an existentially quantified variable y, and a conjunction of Skolem constraints π computed in Algorithm 6.2. Note that AE-VAL generates an exhaustive set of conjunctive predicates that together describe the space of the Skolem predicate (i.e., in Disjunctive Normal Form) and our algorithm is trivially generalizable to this form by separately solving each conjunct and introducing a random choice between them. Algorithm 6.3 constructs a graph of a function that is embedded in a relation, specified by a conjunction of expressions over the relational operators $\{=, \neq, >, \geq, \leq, <\}$, using the following constraints:

$$E \overset{\text{def}}{=} \{y = f_i(x)\}_i \quad D \overset{\text{def}}{=} \{y \neq f_i(x)\}_i \quad G \overset{\text{def}}{=} \{y > f_i(x)\}_i$$
$$GE \overset{\text{def}}{=} \{y \geq f_i(x)\}_i \quad LE \overset{\text{def}}{=} \{y \leq f_i(x)\}_i \quad L \overset{\text{def}}{=} \{y < f_i(x)\}_i$$

In addition to the constraints above, Algorithm 6.3 also utilizes the following helper functions (where $\sim \in \{<, \leq, =, \neq, \geq, >\}$):

$$\mathrm{ASN}(y \sim e(x)) \overset{\mathrm{def}}{=} e \quad \mathrm{MIN}(\{s\}) \overset{\mathrm{def}}{=} \mathrm{ASN}(s) \quad \mathrm{MAX}(\{s\}) \overset{\mathrm{def}}{=} \mathrm{ASN}(s)$$

$$\mathrm{MIN}(S) \overset{\mathrm{def}}{=} \mathit{ite}(\mathrm{ASN}(s)_{(\leq)}\mathrm{MIN}(S\backslash\{s\}), \mathrm{ASN}(s), \mathrm{MIN}(S\backslash\{s\})), s \in S$$

$$\mathrm{MAX}(S) \overset{\mathrm{def}}{=} \mathit{ite}(\mathrm{ASN}(s)_{(\geq)}\mathrm{MAX}(S\backslash\{s\}), \mathrm{ASN}(s), \mathrm{MAX}(S\backslash\{s\})), s \in S$$

Operator MIN (MAX) computes a symbolic minimum (maximum) of the given set of constraints. While the algorithm is applicable for both LIA and LRA, the following transformations are used for the case of integers:

$$\frac{A < B}{A \leq B - 1} \qquad \frac{A \geq B}{A > B - 1}$$

These transformations help avoid clauses containing $<$ and $\geq$. Line 1 initializes the value of the boolean flags ℓ_{closed} and u_{closed} to false, and line 2 handles the case where equality constraints exist over y. Lines 3 to 8 construct the expressions for the lower and upper bounds, and the truth of the flags depends on the (symbolic) comparison between the symbolic minima and maxima. Line 9 handles the case where the lower bound is equal to the upper bound. It should be noted that for cases handled by lines 2 and 9 only deterministic choices exists.

Lines 10 to 14 attempt to compute an expression containing a URNG that considers the set of disequalities D. First, the algorithm extracts the right-hand side of disequalities in line 10. If both bounds are undefined, line 11 returns the application of the URNG $f_{\mathrm{RNG}}(H, \top, \top, -\infty, +\infty)$, where $-\infty$ and $+\infty$ are represented as free variables that can be later mapped respectively to the minimum and maximum arithmetic representations supported by the implementation (e.g. `INT_MIN` and `INT_MAX` for integers in C). If only the lower bound is undefined (line 12), we use $f_{\text{RNG}}(H, \top, u_{closed}, -\infty, u)$ to generate a random value with an unconstrained lower bound. Similarly, we handle the case where no constraints exist for the upper bound in line 13. In line 14, both ℓ and u are defined and the algorithm returns $f_{\text{RNG}}(H, \ell_{closed}, u_{closed}, \ell, u)$ to capture a random value within the respective bounds. In all above cases, when $H \neq \varnothing$, the URNG is expected to generate a value that satisfies all disequality constraints in D. For the special case where $D = \varnothing$, there are no such disequalities over y and the Skolem term can freely assign any value within the computed bounds ℓ and u.

As an illustration of our procedure, we present summarized runs over the following examples.

Example 6.4.1. *Consider the formula $\forall x. \exists y_1, y_2 . \psi(x, y_1, y_2)$ over LIA, where:*

$$\psi(x, y_1, y_2) \overset{\text{def}}{=}$$
$$(x \leq 2 \wedge y_1 > -3x \wedge y_2 < x) \vee (x \geq -1 \wedge y_1 < 5x \wedge y_2 > x)$$

The formula is valid since there exists an assignment to y_1 and y_2 that satisfies the constraints in ψ, for any x. In order to construct such a witness, AE-VAL *needs to consider two separate cases for x, i.e., the constraints $x \leq 2$ and $x \geq -1$.*

Under $x \leq 2$, the deterministic Skolem terms generated by the original Skolemization procedure in AE-VAL *would be $-3x + 1$ for y_1 and $x - 1$ for y_2. For the random case, Algorithm 6.3 computes $f_{\text{RNG},y_1}(\varnothing, \bot, \top, -3x, +\infty)$ and $f_{\text{RNG},y_2}(\varnothing, \top, \bot, -\infty, x)$. Under $x \geq -1$, the deterministic terms would be $-3x + 1$ for y_1 and $x + 1$ for y_2, while Algorithm 6.3 computes the functions $f_{\text{RNG},y_1}(\varnothing, \top, \bot, -\infty, 5x)$ and $f_{\text{RNG},y_2}(\varnothing, \bot, \top, x, +\infty)$,* *respectively.*

For the sake of completion, note that the above terms are finally combined into the Skolem term for $\exists y_1, y_2 . \psi(x, y_1, y_2)$*:*

$$
\begin{aligned}
sk_{\vec{y}}(x) \overset{\text{def}}{=} ite(&x \leq 2,\\
&(y_1 = f_{\text{RNG},y_1}(\varnothing, \bot, \top, -3x, +\infty) \wedge y_2 = f_{\text{RNG},y_2}(\varnothing, \top, \bot, -\infty, x)),\\
&\quad (y_1 = f_{\text{RNG},y_1}(\varnothing, \top, \bot, -\infty, 5x) \wedge y_2 = f_{\text{RNG},y_2}(\varnothing, \bot, \top, x, +\infty)))
\end{aligned}
$$

Example 6.4.2. *Consider an Assume-Guarantee contract for a system with the input vector* $\vec{x} = \{x_1, x_2, x_3\} \in \mathbb{R}^3$ *and one output* $y \in \mathbb{R}$ *and the following constraints*

- $A(x_1, x_2) \overset{\text{def}}{=} x_1, x_2 \in (0, 1)$
- $G_I(y) \overset{\text{def}}{=} \top$
- $G_T(y, x_1, x_2, y') \overset{\text{def}}{=} y' \in (0, 1) \wedge y' \neq x_1 \wedge y' \neq x_2$

The above specification is realizable as there are infinitely many assignments to y *that satisfy the guarantees* G *given any value of* x_1, x_2 *in* $(0, 1)$*. Using Algorithm 6.3 we retrieve the following Skolem term to enable random behavior (note that input* x_3 *is not included in the set* H*, i.e. the first argument of the function):*

$$sk_y(\vec{x}) \overset{\text{def}}{=} f_{\text{RNG},y}(\{x_1, x_2\}, \bot, \bot, 0, 1).$$

Example 6.4.3. *Consider the contract from Figure 6.1. The details of the synthesis procedure remain identical with the deterministic approach up until the Skolemization step. Figure 6.3 shows the C implementation for the random witness that is synthesized using Algorithm 6.3. Our proposed Skolemization procedure returns the assignment value*

```
void skolem() {
  if(position + x == 1) {
    y = RandVal(1, 1, -1, 1);
  } else if (position + x >= -1 &&
        position + x <= 0) {
    y = - (position + x);
  } else {
    y = RandVal(1, 1, -1, 1);
  }
}
```

Figure 6.3: Synthesized random witness.

```
double RandVal(_Bool lflag, _Bool uflag,
  int min = lflag ? lbound : lbound+1;
  int max = uflag ? ubound : ubound-1;
  int range = max - min + 1;
  double rnd = (double) rand() /
        (1.0 + (double) RAND_MAX);
  int value = (int) ((double) range * rnd);
  return value + min;
}
```

Figure 6.4: Example random number generator.

for y that is equivalent to $f_{\text{RNG}}(\top, \top, -1, 1)$*, for the conditions under which the system can safely choose to move the robot either left, right, or not at all. The actual choice is randomly made through the application of a function named RandVal. The implementation of the function is then left to the engineer's discretion (an example is given in Figure 6.4, noting that there is no argument to represent the set of disequalities).*

6.4.3 Soundness and Completeness

In this section we prove that Algorithm 6.3 is sound, and can provide all possible Skolem terms given a conjunction of Skolem constraints $\pi(\vec{x}, y)$. As we noted in the beginning of Section 6.4.2, the Skolem extraction procedure is easily generalized to the case where $\pi(\vec{x}, y)$ contains disjunctions of constraints. As such, the corresponding part of the proofs is omitted from this section.

Theorem 6.4.1 (Soundness of Skolem Extraction)**.** *Assuming that the properties 1-5 from Def. 6.4.1 hold, Algorithm 6.3 returns valid Skolem terms.*

Proof. To prove this statement, it suffices to show that any computed Skolem term $sk_y(\vec{x})$ by Algorithm 6.3 accompanied by the associated postconditions in Def. 6.4.1, implies the input Skolem constraints in $\pi(\vec{x}, y)$. Return lines 2 and 9 in ExtractSK are trivial cases, as they reduce to a simple assignment from equality constraints. Line 11 refers to the case where no bounds have been defined and the computed Skolem term is a URNG that utilizes the unconstrained variables $-\infty$ and $+\infty$ along with postcondition 5 to ensure the choice of an arbitrary value lies within the specified domain. Lines 12 and 13 handle the case where inequalities exist that determine the lower and upper bounds $\ell(\vec{x})$ and $u(\vec{x})$. If the lower bound is undefined, line 12 returns a URNG that is guaranteed to provide a random value between $-\infty$ and u as per postconditions 4 and 5. We prove the soundness of terms provided by line 13 in a similar manner. If both bounds exist, then in line 14 the Skolem term returned is a URNG guaranteed to provide a value within the range specified by $\ell(\vec{x}), u(\vec{x})$, as per postconditions 2-5. $\square$

Theorem 6.4.2 (Completeness of Skolem Extraction)**.** *The Skolem terms generated by Algorithm 6.3 are sufficient to represent all possible witnesses of the conjunctive* $\forall\exists$*-formula in Eq. 3.3.4.*

Proof. It suffices to prove that no weaker set of postconditions pc' (i.e., $pc \Rightarrow pc'$) exists, such that:

$$\forall \vec{x}. pc'(sk(\vec{x})) \Rightarrow \pi(\vec{x}, sk(\vec{x})) \tag{6.2}$$

We prove this by contradiction, assuming that pc' exists whenever Algorithm 6.3 returns. **Lines 2 and 9**. Algorithm 6.3 returns the deterministic assignments ASN(e) and ℓ, for which no weaker postconditions exist.

Line 11. In this case, no bounds have been defined, and postconditions 1 and 5 are used to denote a range with unconstrained bounds $-\infty$ and $+\infty$. Formally, we can simplify postcondition 5 to $pc = true$, for which no weaker postcondition exists. It is also noteworthy to state that weaker postconditions would have to violate at least one disequality in D and postcondition 1.

Line 12. We have $\ell =$ undef, i.e., no constraints exist for the lower bound, and the Skolem term

$$sk(\vec{x}) = f_{\text{RNG}}(H, \top, u_{closed}, min, u)$$

is returned. Depending on whether the upper bound u is closed or not, we have two cases. For brevity, we show the proof for the case where u is closed, and the corresponding case for the open bound follows similar principles.

- When u is closed, the output constraints are simplified to $\pi(\vec{x}, sk(\vec{x})) = sk(\vec{x}) \leq u$ and the Skolem term

  $$sk(\vec{x}) = f_{\text{RNG}}(H, \top, \top, -\infty, u)$$

  is returned, with postcondition 5 capturing the term's range. Assume that a weaker postcondition pc' exists with the same set of disequalities D, such that Eq. 6.2 holds. Without loss of generality, we pick

  $$pc' = f_{\text{RNG}}(H, \top, \top, -\infty, u) \in [-\infty, u']$$

  with $u' > u$. Therefore, we have that $pc \Rightarrow pc'$, but Eq. 6.2 does not hold for pc', as the new term may provide the value u' as an output, falsifying $\pi(\vec{x}, sk(\vec{x}))$.

Line 13. Similar to proof for line 12.

Line 14. $\ell \neq u \neq$ undef, and as such the output constraints can be simplified into

$\pi(\vec{x}, sk(\vec{x})) = \ell \sim sk(\vec{x}) \sim u$, where $\sim\ \in \{<, \leq\}$. We have the following cases corresponding to the possible ranges:

1. (ℓ, u). In this case we have $sk(\vec{x}) = f_{\text{RNG}}(H, \bot, \bot, \ell, u)$ and as postcondition pc the second postcondition from Def 6.4.1. Assume that a weaker postcondition pc' exists, such that Eq. 6.2 holds. We can pick $pc' = f_{\text{RNG}}(H, \bot, \bot, \ell, u) \in [\ell, u]$. In this case, $pc \Rightarrow pc'$ holds, but Eq. 6.2 does not hold, as we can pick any of the assignments $sk(\vec{x}) = \ell$, $sk(\vec{x}) = u$, which violate the constraints in π, reaching a contradiction.

2. $[\ell, u)$. Similar to the previous proof, by picking, e.g., $pc' = f_{\text{RNG}} \in [\ell, u]$.

3. $(\ell, u]$. Similarly, we can pick $pc' = f_{\text{RNG}} \in [\ell, u]$.

4. $[\ell, u]$. Similarly, we can pick $pc' = f_{\text{RNG}} \in [\ell', u]$, where $\ell' < \ell$.

$\square$

6.5 Implementation and Evaluation

We implemented our random synthesis algorithm as a complementary procedure to the original synthesis framework JKind [48], a Java implementation of a popular Kind model checker [14, 48, 63]. Following Kind, the input contracts are expressed using the Lustre dataflow language [61]. JKind provides support for synthesis both through the fixpoint algorithm in JSyn-vg as well as its predecessor, JSyn, a realizability checking algorithm based on the k-induction principle [49, 64, 65]. Our proposed Skolemization procedure in Algorithm 6.3 is a new extraction method that is performed after the validity checking procedure in AE-VAL, thus making it inherently compatible with both JSyn and JSyn-vg. It is noteworthy that our approach does not add any performance overhead to the baseline implementation of JSyn-vg, as shown in Table 6.2.

Since the synthesized Skolem functions are expressed in the SMT-LIB 2.0 language [5] by default, we translate them into executable C implementations. For the purposes of this paper, we mapped the application of URNGs to calls to random number generators of uniformly distributed values, unless otherwise noted.

The evaluation process of our work is twofold:

1. *Empirical.* We performed case studies in applications where synthesis of random designs can be beneficial.[5] For the first case study, we conducted an experiment in the context of model-based fuzz testing, where the goal was to synthesize *reactive graybox fuzzers* capable of exposing vulnerabilities that can crash an application, through random test case generation. The second study revolves around *controller synthesis* for avoidance games in robot motion planning.

2. *Synthesis time.* We investigated the effect that our Skolemization algorithm had on JSyn-vg in terms of synthesis time. Furthermore, we compared our work to DT-Synth, a state-of-the-art synthesis tool for infinite-state problems [79].

6.6 Case Study 1: Reactive Fuzzers

In our first case study, we explored the applicability of synthesized implementations with random behavior in fuzz testing. We focused on model-based approaches to examine a system-under-test (SUT), the input specification of which was used to derive test cases (see Utting et al. for a detailed survey [100]). In the past, model-based fuzz testing revolved around the use of structured descriptions of the system input in the form of grammars and a sophisticated implementation of a fuzzer that, given a grammar, would continuously feed random inputs to the SUT [1, 3, 84, 101]. We show that synthesis offers a viable alternative technique in this context, where the generated implementations

[5]The benchmarks are available at https://figshare.com/s/ce2dfd885b3caf20f46d.

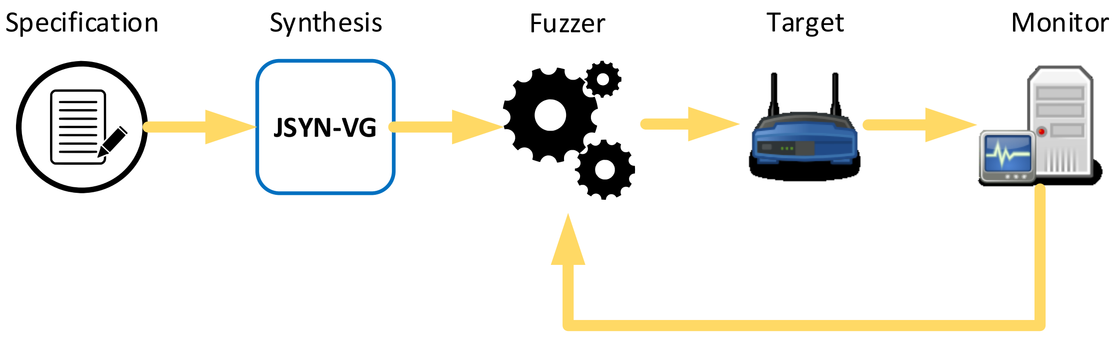


Figure 6.5: Fuzzer synthesis and testing diagram

can serve as SUT-specific fuzzers, requiring for configuration nothing but the input specification for the SUT.

6.6.1 Setup and Evaluation

The main intuition is that the SUT's input description can be viewed as a substantial fragment of the fuzzer's specification, which can then be used to synthesize a reactive random test case generator. Figure 6.5 depicts our exact setup, where the designer already has a specification for the SUT and uses JSYN-VG with our Skolemization algorithm to automatically generate a corresponding fuzzer. The fuzzer is then attached to the SUT (Target), along with an accompanied monitoring service (Monitor) that tracks progress with respect to the SUT-related statistics (e.g., coverage). Following the definition of *graybox fuzzing*, a feedback loop exists where monitored information can be subsequently fed to the fuzzer, in order to dictate the generation of future test cases.

Table 6.1: Fuzzer performance comparison and synthesis times.

	AFL				AFLFast				Synthesized Fuzzer		
	w/o corpus		w/ corpus		w/o corpus		w/ corpus				
System Under Test	**Cvg**	**Cr?**	**Cvg**	**Cr?**	**Cvg**	**Cr?**	**Cvg**	**Cr?**	**Cvg**	**Cr?**	**Synthesis (s)**
basic_messaging	**83.76%**	✘	82.48%	✘	82.48%	✘	82.48%	✘	81.21%	✔	16.067
Dive_Logger	92.18%	✘	92.41%	✘	92.18%	✘	92.18%	✘	**93.79%**	✔	99.852
Divelogger2	80.50%	✘	**87.04%**	✘	78.40%	✘	85.08%	✘	83.25%	✘	77.689
Email_System_2	78.71%	✘	**92.90%**	✘	91.61%	✘	**92.90%**	✘	84.84%	✔	36.624
Movie_Rental_Service	38.32%	✘	38.72%	✘	38.32%	✘	38.72%	✘	**49.50%**	✔	140.826
Palindrome	**78.13%**	✔	**78.13%**	✔	**78.13%**	✔	**78.13%**	✔	75.00%	✔	1.231
PTaaS	71.94%	✔	**78.06%**	✔	46.76%	✔	46.76%	✔	74.46%	✔	77.041
Quadtree_Conways	67.04%	✘	**84.08%**	✘	67.04%	✘	67.04%	✘	64.79%	✘	88.743
SCUBA_Dive_Logging	80.97%	✔	80.67%	✔	76.41%	✔	76.41%	✔	**83.56%**	✔	101.813
User_Manager	58.43%	✘	67.45%	✘	29.02%	✘	29.02%	✘	**79.80%**	✘	16.289

Cvg stands for SUT line coverage, Cr? indicates whether fuzzer crashed the application.

Using this setup, we proceeded with a thorough performance evaluation of our synthesized fuzzers, following guidelines that were recently proposed by Klees et al. [66]:

SUT Selection. We considered ten applications from the DARPA Cyber Grand Challenge (CGC) [6], a benchmark collection that has been extensively used in the past to assess the performance of fuzzers due to the high degree of interactivity between the SUT and the user [83, 89, 96]. The original collection was aimed towards the evaluation of automated reasoning and testing tools, and each application is intentionally documented in a way that is insufficient to derive a precise specification from the documents themselves. To simulate the context under which synthesis would make most sense as a tool, we closely inspected and ran each of the ten applications that appear in this paper, in order to identify the types and sequences of inputs each application takes.

Fuzzer specification. Using the information regarding each application's inputs, we wrote a corresponding Assume-Guarantee contract for a fuzzer. Each fuzzer specification consists of properties that capture the valid ranges of values for each one of the SUT inputs. Moreover, the specification is stateful, making each fuzzer reactive to changes (or lack thereof) in coverage results from previously generated tests. We specified the behavior of the fuzzer in such a way that, for the majority of its runtime, valid inputs are fed into the SUT. When no progress is made in terms of coverage, the fuzzer attempts generating invalid tests with probability $p = 0.2$.

Formalization. All of the aforementioned elements that comprise the fuzzer specification can be expressed using a set of *safety* properties over the SUT inputs, where each set precisely captures the conditions under which a (in)valid value is generated for the corresponding input. An example pair of such properties is the following:

- $\mathit{prop}_1 \overset{\text{def}}{=} p' \geq 0 \land p' \leq 1$
- $\mathit{prop}_2 \overset{\text{def}}{=} (\neg \mathit{cvg} \land (p \leq 0.1 \lor p \geq 0.9)) \Rightarrow \mathit{in}'_{\text{sys}} \notin S_{\text{valid}}$

[6]The public CGC benchmark collection is available at https://bit.ly/2HBqrJq.

Variables p and in_{sys} are fuzzer outputs, with in_{sys} also serving as a corresponding input for the SUT. The value of $p \in [0, 1]$ is picked randomly for each test, and it determines whether the next (primed) system input in'_{sys} will be assigned to a valid value (i.e., a value in S_{valid}) or not. Variable cvg is an input to the fuzzer and can be viewed as a flag which, when set, informs the fuzzer that the previous test resulted in progress in system coverage (e.g., line coverage improved). If such progress was not observed, then we allow the fuzzer to randomly consider invalid values in subsequent tests. More specifically, when $p \leq 0.1 \vee p \geq 0.9$, the fuzzer will generate an invalid value, i.e., a value that does not satisfy the constraints that define S_{valid}. Following the notation that we described in previous sections, the synthesis problem for the properties above is to ensure that $\forall p, cvg, in_{\text{sys}} \exists p', in'_{\text{sys}}.(prop_1 \wedge prop_2)$ is valid.

Synthesis and Evaluation. Using the fuzzer contracts, we synthesized a fuzzer for each application and ran it against the SUT using the setup in Figure 6.5. We set the timeout for each fuzzing campaign to nine hours, and monitored the SUT line coverage (*gcov*) as well as crashes. To compare performance, we also ran fuzzing campaigns using AFL [103] and AFLFast [10], using their default configurations. We selected these tools primarily due to AFL being one of the most prominent tools in the area, while AFLFast is a recent extension to AFL that has been shown to perform better with respect to vulnerability detection.[7] Both tools were run both with and without an initial corpus in order to provide a more complete picture of their performance, whether the user provides additional information or not. To remain fair with respect to the evaluation, the corpora were created using tests that exercise application locations that are as deep as possible.

Table 6.1 shows the results of our experiments. Most of the applications contain

[7]Both AFL and AFLFast do not support line coverage reporting natively. To monitor coverage, we used *afl-cov* [88], a wrapper tool that enables the use of *gcov* with AFL and its variants.

unreachable code related to debugging methods, and as such 100% coverage is not attainable using *gcov* without further modifications to the source code. While we were able to achieve $\geq$ 75% line coverage for the majority of the benchmarks, the application "Movie_Rental_Service" was the worst performing with only 49.5%. Despite that, the synthesized fuzzer outperformed both AFL and AFLFast on either configuration with a significant margin. In fact, our synthesized fuzzers outperformed both AFL and AFLFast on four applications and remained within 4% of the best performing tool for five others, with "Quadtree_Conways" being the only exception. More interestingly, seven of the synthesized fuzzers were able to crash the corresponding application at least once, whereas AFL/AFLFast were only able to crash three.

Considering the performance results along with the low synthesis time per fuzzer, we believe that synthesis of model-based fuzzers should be considered a viable tactic towards testing systems where a specification already exists. Arguably, a synthesized fuzzer is as easy to use as a general-purpose tool like AFL. Furthermore, the user does not have to provide additional information through a corpus, a procedure in testing that often times can be time consuming and cumbersome, as both valid and invalid input sequences have to be considered for a successful campaign.

6.7 Case Study 2: Robot Motion Planning

In our second study, we synthesized implementations for robots participating in two-player safety games against an adversary. The study is furthermore split into two parts.

6.7.1 Simulating avoidance games

We experimented on simulating an avoidance game in a bounded arena, where the synthesized solution was used against two different adversarial scenarios. Both the properties of the robot and the adversary were specified using their position in terms of (x, y)

coordinates. Formally, we described the game using the following properties:

- Initial state : The robot starts in $(x_{\text{init}}, y_{\text{init}})$ (similarly for the adversary).
- Valid transitions : $x'_{\text{robot}} \in [x_{\text{robot}} - \delta, x_{\text{robot}} + \delta]$, where δ is user-defined and captures the maximum distance between subsequent moves (similarly for y-coordinate and the adversarial transitions).
- In-bounds property : $x_{\text{robot}} \geq x_{\text{min}} \wedge x_{\text{robot}} \leq x_{\text{max}}$ (similarly for the y-coordinate).
- Avoidance property : $x_{\text{robot}} \neq x_{\text{adversary}} \vee y_{\text{robot}} \neq y_{\text{adversary}}$.

The first scenario in our presentation involves the adversary patrolling on a specific route, while in the second the adversary is always moving towards the robot. Trajectory videos for both scenarios are available online[8].

Real Coordinates.

Figure 6.6 shows three possible trajectories that were generated after running the synthesized solution for 1000 turns against the patrolling adversary. Both robots move in the arena using rational coordinates in a 5x5 box. The initial location for the robot is the point $(0.5, 0.5)$ and the adversary begins its route from $(0.8, 0.8)$. While the adversary has a predetermined route, the robot is allowed to move towards any possible direction (vertically, horizontally and diagonally). Moreover, the robot can move at varying distances up to 0.1 units away from its current position, in both axes (i.e., $|x_{\text{robot}} - x'_{\text{robot}}| \leq 0.1$, and similarly for the y axis). Figure 6.6a indicates how the synthesized solution can respond in a random pattern, covering different parts of the bounded arena while preserving safety. Figure 6.6b and 6.6c demonstrate the resulting

[8]Pictures and videos of the simulated games presented in this section were anonymized and made available at https://figshare.com/s/ce2dfd885b3caf20f46d.

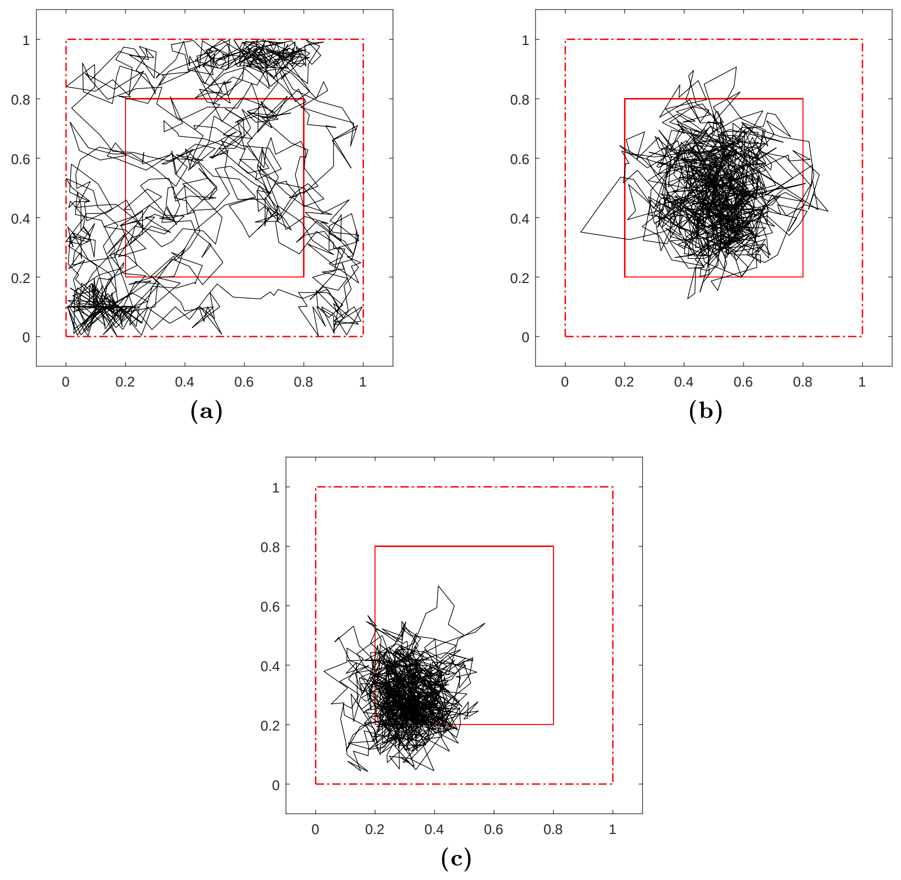


Figure 6.6: Random trajectories of a robot (irregular solid line) while avoiding a patrolling adversary (inner square).

trajectories when the user introduces bias in the values returned by the random number generators, using the same generated witness from AE-VAL. As a result, the robot was limited to moves that would retain its position within the central area of the arena (Figure 6.6b) and close to the bottom left corner of the patrolling adversary's route (Figure 6.6c).

Integer Coordinates.

For this experiment, we aimed to demonstrate the advantages that randomness can provide with respect to how well a robot covers a bounded arena, inspired by work in coverage path planning problems. Figure 6.7 shows how two trajectories evolved over several turns (100, 250 and 1000 turns) for a similar motion planning problem using integer coordinates. To demonstrate which parts of the arena the robot explored we outline its trajectory with a bold black line, while the red line represents the trajectory of the adversary. In this game, the adversary is aggressively chasing after the robot in a random fashion. The robot's objective remains the same, i.e., move within the bounded arena while avoiding the adversary. The robot's initial location is the point $(0, 0)$, while the adversary begins at $(6, 6)$.

In fact, Figure 6.7a, 6.7c, and 6.7e show moves performed by a random controller, while Figure 6.7b, 6.7d,and 6.7f depict the behavior of the deterministic solution provided by the standard synthesis algorithm in JSYN-VG. It is apparent that the former visits 100% states in less than 250 turns, whereas the latter visits only 30% states in 1000 turns. This comparison showcases the advantages that a random solution can provide in terms of overall coverage as well as the diversity of behaviors that can be observed and exercised when an implementation can be generated that always considers the entire set of safe choices, instead of an instantiated strategy.

6.8 Evaluation – Synthesis time

Our case study in robot motion planning was inspired by results in this context from the most recent and related work on DT-SYNTH [79]. This reactive synthesis framework incorporates learning techniques to generate winning sets for infinite-state safety games in the form of decision trees. DT-SYNTH has been shown to outperform previous proposed synthesis tools, both in infinite-state (ConSynth [6] and finite-state problems (RPNI-Synth and SAT-Synth [80]). While the authors do not explicitly talk about randomness, the winning sets provided by DT-Synth are sufficient to generate implementations with diverse behavior. Despite this fact, the generated winning sets are subsets of the greatest fixpoint of safe states, which would lead to implementations that only exercise a fragment of the reachable state space. An example is the winning set that we mentioned for the motivating example in Section 6.3.

Note that DT-Synth works only for finite-branching game graphs, and the user must additionally specify a minimum value for the number of successors for each vertex in the graph. An incorrect value for this threshold can lead to unsound witnesses. With our JSyn-vg, such additional knowledge is not required from the user since it is only reliant on the original specification and is guaranteed to provide sound results, thanks to Theorem 6.4.1.

Table 6.2 presents the comparison of JSyn-vg and DT-Synth. As an addendum we included the synthesis times for the problems using the existing deterministic synthesis algorithm in JSyn-vg. As we mentioned in Section 6.5, the performance is identical when compared to synthesizing random witnesses.

For the purposes of this comparison, we used the infinite-state benchmarks presented in the original paper on DT-Synth [79], as well as the simulated avoidance games from Section 6.7.1, namely *bounded_evasion* and *bounded_evasion_ints*. The two tools have similar performance for half the benchmarks, with significant differences for the rest. For *diagonal*, the main distinction is that DT-Synth requires the definition of two additional expressions to guide the learning procedure, whereas JSyn-vg finds a solution without additional templates. On the other hand, DT-Synth's ability to synthesize memoryless strategies allows for faster synthesis for *solitarybox*, where the robot is simply moving freely within an infinite arena while staying within a horizontal stripe of width equal

Table 6.2: Synthesis time of DT-Synth and JSyn-vg (seconds).

Benchmark	**JSyn-vg**	**JSyn-vg (random)**	**DT-Synth**
box	0.603	0.606	**0.258**
diagonal	1.109	**1.011**	6.027
evasion	0.705	0.605	0.660
follow	3.34	3.029	**1.034**
limitedbox	3.229	3.332	3.350
solitarybox	1.902	1.816	**0.284**
square	5.823	**5.605**	6.44
program-repair	3.122	3.638	**2.452**
repair-critical	83.891	88.073	**30.593**
synth-synchronization	23.013	**23.2**	89.804
cinderella ($c = 2$)	20.061	**20.167**	> 900
cinderella ($c = 3$)	12.02	**11.294**	> 900
bounded_evasion	49.528	**49.662**	unsupported
bounded_evasion_ints	31.614	**32.611**	> 900

to three. JSyn-vg is targeted at synthesis of stateful systems, and as such, a more elaborate strategy is generated.

In the case of *repair-critical*, DT-Synth appears to be more efficient in terms of handling disjunctive expressions in the specification, while for *synth-synchronization* DT-Synth seems to require more elaborate hypotheses in order to come up with a witness. The latter is further demonstrated in the results for the *cinderella* and *bounded_evasion_ints* games, where DT-Synth fails to synthesize a witness within the timeout of 15 minutes. In contrast, JSyn-vg computes a greatest fixpoint of safe states and synthesizes a solution in a few seconds. Finally, for the game *bounded_evasion*, DT-Synth does not currently support the theory of linear real arithmetic.

6.9 Related Work

The idea of synthesizing reactive designs with random behavior is relevant to synthesis for permissive games. This area has been explored in the past for finite-state problems [12, 69]. More recently, Fremont and Seshia described a formal extension to the theory of Control Improvisation to support reactive synthesis [46]. Their probabilistic approach is limited to finite-state problems and practically useful only for the subset of safety games. The end result is a maximally-randomized finite word generator, called an improviser, where each word satisfies the predetermined probability threshold constraints. In comparison, our approach synthesizes designs that simulate randomness for infinite-state problems. Furthermore, we do not provide guarantees regarding the randomness of the responses from the synthesized witness. Instead, we focus on synthesizing witnesses that consider regions of values as candidates to variable assignments, a problem reducible to SMT. In our case, the end product is an implementation that can be further refined by the engineer with a custom probability distribution to retrieve random values. This provides and added degree of freedom as the user can then choose whether to express bias through the requirements or through the random number generators themselves.

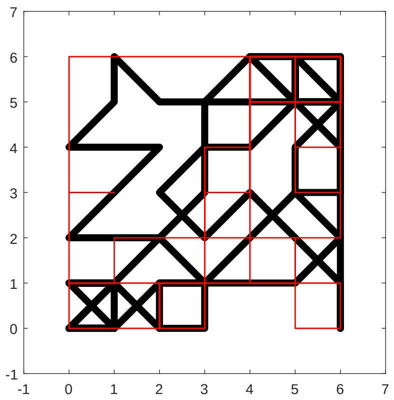
The original work on JSyn-vg targeted the area of infinite-state problems. In this context, Beyenne et al. first proposed a template-based approach called ConSynth, where the specification is accompanied by a template regarding the shape of the solution to guide the synthesizer towards a solution [7]. In contrast, JSyn-vg is a completely automated approach exempting the user from necessity to further reason about the shape of the computed solution and allowing to focus on expressing the problem in the form of input-output contracts. Permissive solutions for infinite-state games have primarily been proposed in the context supervisor synthesis [23, 94], where a controller is synthesized considering a formal representation of the behavior (inputs) provided by the participating plant. Compared to this work, our proposed solution explores the applicability of synthesized controllers with random behavior, while the overall synthesis task is inherently harder due to not requiring an exact model of the controller's environment.

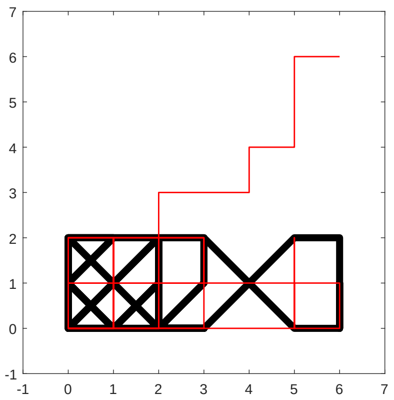
Neider and Markgraf recently proposed DT-Synth [79], a learning-based approach to synthesizing winning sets for infinite state games in the form of decision trees, as an extension to previous work by Neider and Topcu for finite-state problems [80]. DT-Synth requires additional knowledge regarding the number of successor states, where lack thereof can lead to unsound results. As with ConSynth, for more complex problems, the game specification is supported by additional syntactic expressions that help the learner converge faster to a solution. In contrast, our Skolem extraction algorithm is guaranteed to provide sound witnesses and does not depend on additional user-provided input.

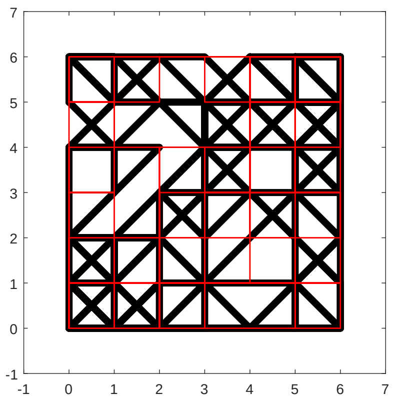
6.10 Conclusion

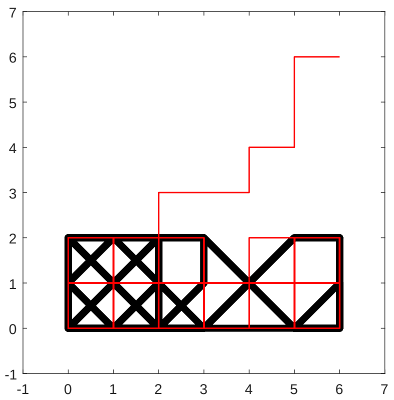
We have presented a novel Skolemization procedure for the AE-VAL solver enabling the synthesis of infinite-state reactive implementations with random behavior. The proposed solution is an extension to AE-VAL and the synthesis algorithm JSyn-vg that computes a greatest fixpoint of safe states. The product is a witness where values inside safe regions are considered as equally safe candidate assignments. Our solution provides the engineer with flexibility when it comes to introducing additional bias through the specification or the implemented random number generators.

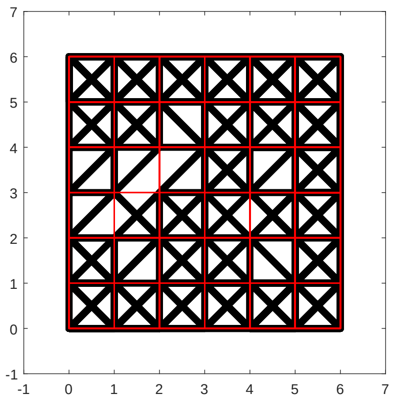
To the best of our knowledge, this is the first work that is capable of synthesizing infinite-state systems with random, specification-compliant behavior. We showed how the extended synthesis framework can be effectively used to synthesize promising solutions in the context of robot motion planning, as well as a novel application in fuzz testing. In the future, we wish to continue exploring the area of reactive fuzzer synthesis, particularly in the context of identifying a formal specification standard. To expand its applicability in robot motion planning, we wish to explore ways to support liveness specifications, as well as soft requirements. The outstanding result of this work is a Skolem extraction procedure general enough to be applicable to other, unexplored aspects of the synthesis problem.


(a) 100 turns


(b) 100 turns


(c) 250 turns


(d) 250 turns


(e) 1000 turns

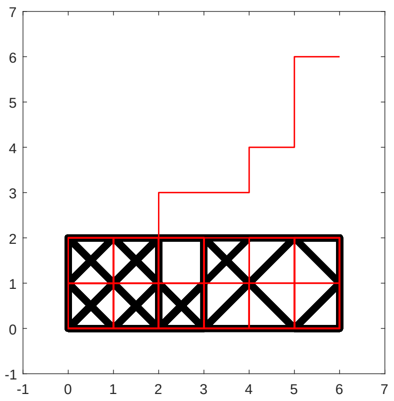

(f) 1000 turns

Figure 6.7: Trajectories and area coverage over time of random (left column) versus deterministic (right column) controller.

Chapter 7

Summary and Future Work

In this dissertation we presented novel techniques towards the realizability checking and synthesis of infinite-state reactive systems. The following subsections summarize the contributions introduced in this work, and discuss future research avenues to explore.

7.1 Thesis Summary

Chapter 4 presented the first ever application of k-induction in the context of reactive synthesis, and demonstrated its usefulness in industrial-level scenarios. K-induction has been primarily popular in model checking in the past, and we demonstrated how it can be used towards other aspects in the formal analysis of requirements. We provided a rigorous proof of its soundness with respect to realizable contracts and created a machine-checked proof using Coq to gain high assurance in the correctness of the formulation and our results.

While the k-inductive realizability checking algorithm was is sound with regards to realizable results, the approximations that were used in practice could not guarantee soundness when the algorithm declares a specification as unrealizable. Chapter 5 presented our efforts to tackle the soundness issue, pursuing a different approach towards solving the problem of synthesis. In this chapter, we presented a new synthesis algorithm based on the mathematical concept of greatest fixpoints. Intuitively, the algorithm computes the greatest set of system states that can be considered towards synthesis of safe implementations. Our experimental comparisons showed how the fixpoint algorithm can outperform k-induction when synthesis is considered and, most importantly, how it is universally sound. A direct comparison depicted that our approach was competitive with, and in many cases outperformed, other state of the art tools.

Finally, Chapter 6 presents new, previously unexplored applications of synthesis when the generated implementations can exhibit a diversified portfolio of possible behaviors. Diversity can be directly related to the range of safe assignments that the synthesized witness can consider when reacting to certain conditions. This chapter proposed a novel Skolemization algorithm that enables synthesis of such systems, reducing the problem to computing the precise set of safe states under environmental assumptions. The outcome is an executable implementation that utilizes randomness as a tool towards picking a random safe state for its next response. Through empirical evaluation, we showcased how these implementations can be valuable in applications in robot motion planning and model-based fuzz testing, where synthesis has never before been considered as a potential solution.

7.2 Future Research Directions

The results presented in this dissertation pose exciting future research directions. In this section, we list the ones that we believe are of significant interest.

Supporting liveness specifications. The algorithms presented in this thesis are limited to specification that admits safety properties. As such, requirements describing that "something good must eventually happen" are not currently supported. While liveness is arguably not as important in safety-critical applications, they are still useful to describe agent goals in robot motion planning problems. In combination with the fact that artificial intelligence has seen an unprecedented blowup in popularity due to advancements in autonomy, it is reasonable to expect that the formal synthesis of systems with liveness properties will be relevant for many years to come. Currently, the possibility of supporting liveness in our proposed synthesis framework remains unknown. Despite this, we believe that the direction that yields the biggest potential is in the greatest-fixpoint approach to solving synthesis. Intuitively, it may be possible to utilize the intermediate candidate fixpoints computed by JSyn-vg, in order to identify a looping sequence of viable states that are guaranteed to lead to the eventual satisfaction of the liveness property. In practice, this would translate to direct performance overhead over the current support for safety properties and as such, development of techniques to alleviate this issue is an equally crucial research goal.

Debugging unrealizable specifications. While synthesis tools are traditionally evaluated in terms of their ability to generate implementations for complex problems, understanding unrealizability is the "other side" of the synthesis "coin" that arguably stands at the same level in terms of research significance, as it is an invaluable asset towards efficient debugging of erroneous specifications. This context has been explored in the past, primarily through the proposal of techniques that identify minimal sources of unrealizability within a contract [22, 70, 71]. While minimality in explanations is important, it comes with severe overhead both in terms of scalability, as well as the ability for the engineer to actually understand the artifacts that are produced by these formal techniques. We believe that there exists a big opportunity in further exploring this area as realizability, under certain conditions, can be solved using divide-and-conquer tactics that preserve soundness of results. Furthermore, at the user interface level, we believe that advancements in modern graphical interfaces can be utilized to present results related to unrealizable specifications in a clear and unambiguous way. The latter applies also in the context of automated repair of unrealizable specifications [75], where the problem of choosing between a number of suggested repairs remains unclear.

Synthesis of designs with random behavior. In Chapter 6, we presented an extensive empirical study on the applicability of synthesized witnesses that are capable of exercising diverse sets of behaviors. Since this is an aspect of synthesis that has not been considered in the past, we believe that there exists considerable research potential in exploring its applicability to similar problems. In robot motion planning, randomness of behavior is an expected feature in order for agents to perform well with respect to coverage path planning problems [50]. On the other hand, synthesis in fuzz testing has never been considered before, despite the fact that model-based fuzzing tools already exist [1, 3, 84, 101]. This fact entertains the question of whether one can identify common patterns in the fuzzer specification that lead to better testing results (SUT coverage, vulnerability detection) over a specific application domain, or properties that can translate to better performance for general-purpose fuzzers. Furthermore, we believe that it would be of interest to explore the applicability of model-based fuzzer synthesis in combination with other formal techniques in testing, such as dynamic invariant detection [33, 34]. In this context, valuable details can be inferred through the generated invariants that could potentially help to guide the fuzzer towards previously uncovered portions of the SUT state space.

Bibliography

[1] Peach fuzzer. https://www.peach.tech/.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2013.

[3] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[4] A. Aziz, F. Balarin, R. Braton, and A. Sangiovanni-Vincentelli. Sequential Synthesis using SIS. *ICCAD*, pages 612–617, 1995.

[5] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[6] T. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, pages 221–233. ACM, 2014.

[7] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, pages 221–234. ACM, 2014.

[8] N. Bjørner and M. Janota. Playing with quantified satisfaction. In *LPAR (short papers)*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015.

[9] M. H. L. Bodlaender, C. A. Hurkens, V. J. Kusters, F. Staals, G. J. Woeginger, and H. Zantema. Cinderella versus the wicked stepmother. In *IFIP TCS*, volume 7604 of *LNCS*, pages 57–71. Springer, 2012.

[10] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on S oftware Engineering*, 45(5):489–506, 2017.

[11] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J. Raskin. Acacia+, a tool for LTL Synthesis. In *CAV*, pages 652–657, 2012.

[12] P. Bouyer, M. Duflot, N. Markey, and G. Renault. Measuring permissivity in finite games. In *International Conference on Concurrency Theory*, pages 196–210. Springer, 2009.

[13] A. R. Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[14] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The kind 2 model checker. In *International Conference on Computer Aided Verification*, pages 510–517. Springer, 2016.

[15] K. Chatterjee and T. A. Henzinger. Assume-Guarantee Synthesis. *TACAS*, pages 261–275, 2007.

[16] B. H. Cheng and J. M. Atlee. Research directions in requirements engineering. In *Future of Software Engineering (FOSE'07)*, pages 285–303. IEEE, 2007.

[17] C.-H. Cheng, Y. Hamza, and H. Ruess. Structural synthesis for gxw specifications. In *CAV*, pages 95–117. Springer, 2016.

[18] H. Choset. Coverage for robotics–a survey of recent results. *Annals of mathematics and artificial intelligence*, 31(1-4):113–126, 2001.

[19] A. Church. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*, pages 23–35, 1962.

[20] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Parameter synthesis with IC3. In *FMCAD*, pages 165–168. IEEE, 2013.

[21] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. HyComp: An SMT-based model checker for hybrid systems. In *TACAS*, volume 9035 of *LNCS*, pages 52–67. Springer, 2015.

[22] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 52–67. Springer, 2008.

[23] K. Claessen, J. Kilhamn, L. Kovács, and B. Lennartson. A supervisory control algorithm based on property-directed reachability. In *Haifa Verification Conference*, pages 115–130. Springer, 2017.

[24] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 2018.

[25] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In *NFM*, pages 126–140. Springer-Verlag, 2012.

[26] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL'85*, pages 151–184. Springer, 1985.

[27] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340. Springer, 2008.

[28] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*, pages 443–456. ACM, 2013.

[29] A. Donzé, R. Valle, I. Akkaya, S. Libkind, S. A. Seshia, and D. Wessel. Machine improvisation with formal specifications. In *ICMC*, 2014.

[30] V. D'silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[31] N. Een, A. Mishchenko, and R. Brayton. Efficient Implementation of Property Directed Reachability. In *FMCAD*, pages 125–134. IEEE, 2011.

[32] R. Ehlers. Symbolic bounded synthesis. In *CAV*, pages 365–379. Springer, 2010.

[33] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[34] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[35] G. Fedyukovich and A. Gupta. Functional Synthesis with Examples. In *CP*, volume 11802 of *LNCS*, pages 547–564. Springer, 2019.

[36] G. Fedyukovich, A. Gurfinkel, and A. Gupta. Lazy but Effective Functional Synthesis. In *VMCAI*, volume 11388 of *LNCS*, pages 92–113. Springer, 2019.

[37] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Automated Discovery of Simulation Between Programs. In *LPAR*, volume 9450 of *LNCS*, pages 606–621. Springer, 2015.

[38] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Automated discovery of simulation between programs. In *LPAR*, volume 9450 of *LNCS*, pages 606–621. Springer, 2015.

[39] B. Finkbeiner. Synthesis of reactive systems. *Dependable Software Systems Engineering*, 45:72–98, 2016.

[40] M. Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[41] P. Flener and D. Partridge. Inductive programming. *Autom. Softw. Eng.*, 8(2):131–137, 2001.

[42] R. L. Flood and M. C. Jackson. *Critical systems thinking*. Springer, 1991.

[43] D. Fraze. The DARPA Cyber Grand Challenge. https://www.darpa.mil/program/cyber-grand-challenge.

[44] D. J. Fremont, A. Donzé, and S. A. Seshia. Control improvisation. *arXiv preprint arXiv:1704.06319*, 2017.

[45] D. J. Fremont and S. A. Seshia. Reactive control improvisation. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 307–326, 2018.

[46] D. J. Fremont and S. A. Seshia. Reactive control improvisation. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification*, pages 307–326, Cham, 2018. Springer International Publishing.

[47] A. Gacek. JKind – an infinite-state model checker for safety properties in Lustre. http://loonwerks.com/tools/jkind.html, 2016.

[48] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The jk ind model checker. In *International Conference on Computer Aided Verification*, pages 20–27. Springer, 2018.

[49] A. Gacek, A. Katis, M. W. Whalen, J. Backes, and D. Cofer. Towards Realizability Checking of Contracts Using Theories. In *NFM*, volume 9058 of *LNCS*, pages 173–187. Springer, 2015.

[50] E. Galceran and M. Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous systems*, 61(12):1258–1276, 2013.

[51] S. Gulwani. Dimensions in program synthesis. In *PPDP*, pages 13–24. ACM, 2010.

[52] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A Reference model for Requirements and Specifications. *IEEE Software*, 17(3):37–43, May 2000.

[53] A. Gupta. Formal hardware verification methods: A survey. In *Computer-Aided Verification*, pages 5–92. Springer, 1992.

[54] G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach.* PhD thesis, University of Iowa, December 2008.

[55] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *FMCAD*, pages 1–9. IEEE, 2008.

[56] J. Hamza, B. Jobstmann, and V. Kuncak. Synthesis for Regular Specifications over Unbounded Domains. *FMCAD*, pages 101–109, 2010.

[57] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, 1985.

[58] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 445–458, 2012.

[59] G. J. Holzmann and R. Joshi. Model-driven software verification. In *International SPIN Workshop on Model Checking of Software*, pages 76–91. Springer, 2004.

[60] F. Howar and B. Steffen. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 123–148. Springer, 2018.

[61] E. Jahier, P. Raymond, and N. Halbwachs. The Lustre V6 Reference Manual.

[62] A. Katis, G. Fedyukovich, A. Gacek, J. D. Backes, A. Gurfinkel, and M. W. Whalen. Synthesis from Assume-Guarantee Contracts using Skolemized Proofs of Realizability. *CoRR*, abs/1610.05867, 2016.

[63] A. Katis, G. Fedyukovich, H. Guo, A. Gacek, J. Backes, A. Gurfinkel, and M. W. Whalen. Validity-guided synthesis of reactive systems from assume-guarantee contracts. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 176–193. Springer, 2018.

[64] A. Katis, A. Gacek, and M. W. Whalen. Machine-checked proofs for realizability checking algorithms. In *VSTTE*, pages 110–123. Springer, 2015.

[65] A. Katis, A. Gacek, and M. W. Whalen. Towards synthesis from assume-guarantee contracts involving infinite theories: a preliminary report. In *FormaliSE*, pages 36–41. IEEE, 2016.

[66] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[67] U. Klein and A. Pnueli. Revisiting Synthesis of GR(1) Specifications. *HVC*, pages 161–181, 2010.

[68] A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34. Springer, 2014.

[69] B. Könighofer, M. Alshiekh, R. Bloem, L. Humphrey, R. Könighofer, U. Topcu, and C. Wang. Shield synthesis. *Formal Methods in System Design*, 51(2):332–361, 2017.

[70] R. Könighofer, G. Hofferek, and R. Bloem. Debugging unrealizable specifications with model-based diagnosis. In *Haifa Verification Conference*, pages 29–45. Springer, 2010.

[71] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *International journal on software tools for technology transfer*, 15(5-6):563–583, 2013.

[72] N. Lee. DARPA's Cyber Grand Challenge (2014–2016). In *Counterterrorism and Cybersecurity*, pages 429–456. Springer, 2015.

[73] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[74] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[75] S. Maoz, J. O. Ringert, and R. Shalom. Symbolic repairs for gr (1) specifications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1016–1026. IEEE, 2019.

[76] B. Monmege, T. Brihaye, M. Estiévenart, H.-M. Ho, G. G. ULB, and N. Sznajder. Real-time synthesis is hard! In *FORMATS*, volume 9884, page 105. Springer, 2016.

[77] A. Murugesan, O. Sokolsky, S. Rayadurgam, M. Whalen, M. Heimdahl, and I. Lee. Linking Abstract Analysis to Concrete Design: A Hierarchical Approach to Verify Medical CPS Safety. *Proceedings of ICCPS'14*, April 2014.

[78] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[79] D. Neider and O. Markgraf. Learning-based synthesis of safety controllers. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 120–128. IEEE, 2019.

[80] D. Neider and U. Topcu. An automaton learning approach to solving safety games over infinite graphs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 204–221. Springer, 2016.

[81] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[82] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[83] H. Peng, Y. Shoshitaishvili, and M. Payer. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[84] V.-T. Pham, M. Böhme, and A. Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 543–553, 2016.

[85] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. In *VMCAI*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.

[86] 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*, pages 179–190. ACM, 1989.

[87] M. Preiner, A. Niemetz, and A. Biere. Counterexample-guided model synthesis. In *TACAS*, pages 264–280. Springer, 2017.

[88] M. Rash. afl-cov - AFL Fuzzing Code Coverage. https://github.com/mrash/afl-cov.

[89] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[90] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *CAV, Part II*, volume 9207 of *LNCS*, pages 198–216. Springer, 2015.

[91] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 73–86. ACM, 2009.

[92] L. Ryzhyk and A. Walker. Developing a practical reactive synthesis tool: Experience and lessons learned. *arXiv preprint arXiv:1611.07624*, 2016.

[93] SAE-AS5506. Architecture analysis and design language, Nov 2004.

[94] M. R. Shoaei, L. Kovács, and B. Lennartson. Supervisory control of discrete-event systems via ic3. In *Haifa Verification Conference*, pages 252–266. Springer, 2014.

[95] S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.

[96] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[97] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, 8.4 edition, 2012-2014.

[98] J. G. Thistle and W. M. Wonham. Control of infinite behavior of finite automata. *SIAM Journal on Control and Optimization*, 32(4):1075–1097, 1994.

[99] S. Tini and A. Maggiolo-Schettini. Compositional Synthesis of Generalized Mealy Machines. *Fundamenta Informaticae*, 60(1-4):367–382, 2003.

[100] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.

[101] S. Veggalam, S. Rawat, I. Haller, and H. Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[102] A. Walker and L. Ryzhyk. Predicate abstraction for reactive synthesis. In *FMCAD*, pages 219–226. IEEE, 2014.

[103] M. Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

Appendix A

Coq Proof Definitions on K-inductive Realizability Checking Algorithms

```
Definition admit {T: Type} : T.

Inductive id : Type :=
Id : nat → id.

Inductive inputs : Type :=
input : id → nat → inputs.

Inductive state : Type :=
st : id → nat → state.

Definition initial := state → Prop.

Definition transition := state → inputs → state → Prop.

Definition iguarantee := state → Prop.
Definition tguarantee := state → inputs → state → Prop.

Definition assumption := state → inputs → Prop.

Inductive reachable (s:state) (init: initial) (t: transition) (a:assumption) : Prop
:=
rch : ((init s) ∨ (∃ (s':state) (inp:inputs),
                                    (reachable s' init t a) ∧ (a s' inp) ∧ (t s' inp s))) →
        reachable s init t a.

Inductive realization (init : initial) (t : transition) (a:assumption) (gi:iguarantee)
(gt: tguarantee) : Prop :=
real : ((∀ (s:state), (init s) → (gi s)) ∧
        (∀ (s s':state) (inp : inputs),
            ((reachable s init t a) ∧ (a s inp) ∧ (t s inp s')) → gt s inp s') ∧
        (∃ (s:state), init s) ∧
        (∀ (s:state) (inp:inputs), (reachable s init t a ∧ (a s inp)) →
           (∃ (s':state), t s inp s')))->
        realization init t a gi gt.

Inductive realizable_contract (a:assumption) (gi: iguarantee) (gt: tguarantee) :
Prop :=
rc : (∃ (init : initial) (t : transition), realization init t a gi gt) → realizable_contract a
gi gt.

CoInductive viable (s:state) (a:assumption) (gi:iguarantee) (gt: tguarantee) :
Prop :=
vbl : (∀ (i:inputs), (a s i) → (∃ (s':state), gt s i s' ∧ viable s' a gi gt)) → viable s a
gi gt.

Inductive realizable (a:assumption) (gi : iguarantee) (gt:tguarantee) : Prop :=
```

rl : (∃ (*s*:*state*), *gi s* ∧ *viable s a gi gt*) → *realizable a gi gt*.

Lemma *init_reachable* : ∀ (*s*:*state*) (*init*:*initial*) (*t*:*transition*) (*a*:assumption), *init s* → *reachable s init t a*.

Lemma *reachable_viable* : ∀ (*s* :*state*) (*init*:*initial*) (*t*:*transition*) (*a*:assumption) (*gi*:*iguarantee*) (*gt*:*tguarantee*),
realization init t a gi gt → *reachable s init t a* → *viable s a gi gt*.

Theorem *realcontract_implies_real* (*init*:*initial*) (*t*:*transition*) : ∀ (*a* : assumption) (*gi* : *iguarantee*) (*gt* : *tguarantee*),
realizable_contract a gi gt → *realizable a gi gt*.

Theorem *reach_via* : ∀ (*s s0*:*state*) (*a*:assumption) (*gi*:*iguarantee*) (*gt*:*tguarantee*),
viable s0 a gi gt → *reachable s* (fun *s* ⇒ *s* = *s0*) (fun *s i s'* ⇒ *gt s i s'* ∧ *viable s' a gi gt*) *a* → *viable s a gi gt*.

Theorem *real_implies_realcontract* (*init*:*initial*) (*t*:*transition*) : ∀ (*a* : assumption) (*gi* : *iguarantee*) (*gt* : *tguarantee*),
realizable a gi gt → *realizable_contract a gi gt*.

```
Fixpoint finite_viable (n:nat) (s:state) (a:assumption) (gt:tguarantee) : Prop :=
  match n with
  | O ⇒ True
  | S n' ⇒ ∀ i, a s i → ∃ s', gt s i s' ∧ finite_viable n' s' a gt
  end.
```

```
Fixpoint extendable (n:nat) (s:state) (a:assumption) (gt:tguarantee) : Prop :=
  match n with
  | O ⇒ ∀ i, a s i → ∃ (s':state), gt s i s'
  | S n' ⇒ ∀ i s', a s i → gt s i s' → extendable n' s' a gt
  end.
```

Definition $\mathit{Basecheck}$ $(n\mathord{:}\mathit{nat})$ $(a\mathord{:}$assumption$)$ $(\mathit{gi}\mathord{:}\mathit{iguarantee})$ $(\mathit{gt}\mathord{:}\mathit{tguarantee})$ $:=$

$\exists\ (s\mathord{:}\mathit{state}),\ (\mathit{gi}\ s \wedge \mathit{finite\_viable}\ n\ s\ a\ \mathit{gt}).$

Definition $\mathit{Extendcheck}$ $(n\mathord{:}\mathit{nat})$ $(a\mathord{:}$assumption$)$ $(\mathit{gt}\mathord{:}\mathit{tguarantee})$ $:=$

$\forall\ s\ a\ \mathit{gt},\ \mathit{extendable}\ n\ s\ a\ \mathit{gt}.$

Lemma $\mathit{viable\_implies\_finite\_viable}$: $\forall\ a\ \mathit{gi}\ \mathit{gt}\ n\ s,$

$\mathit{viable}\ s\ a\ \mathit{gi}\ \mathit{gt} \to \mathit{finite\_viable}\ n\ s\ a\ \mathit{gt}.$

Theorem $\mathit{unrealizable\_soundness}$: $\forall\ (s\mathord{:}\mathit{state})\ (\mathit{init}\mathord{:}\mathit{initial})\ (t\mathord{:}\mathit{transition})\ a\ \mathit{gi}\ \mathit{gt},$

$(\exists\ n, \neg \mathit{Basecheck}\ n\ a\ \mathit{gi}\ \mathit{gt}) \to \neg\ \mathit{realizable\_contract}\ a\ \mathit{gi}\ \mathit{gt}.$

Lemma $\mathit{finite\_viable\_plus\_one}$: $\forall\ n\ a\ \mathit{gt}\ (\mathit{gi}\mathord{:}\mathit{iguarantee})\ (i\mathord{:}\mathit{inputs})\ s,$

$(\mathit{extendable}\ n\ s\ a\ \mathit{gt} \wedge \mathit{finite\_viable}\ n\ s\ a\ \mathit{gt}) \to \mathit{finite\_viable}\ (S\ n)\ s\ a\ \mathit{gt}.$

Lemma $\mathit{extend\_viable\_shift}$: $\forall\ (n : \mathit{nat})\ (a :$ assumption$)\ (\mathit{gi} : \mathit{iguarantee})\ (\mathit{gt} : \mathit{tguarantee})\ (s : \mathit{state})\ (i : \mathit{inputs}),$

$(\mathit{extendable}\ n\ s\ a\ \mathit{gt} \wedge \mathit{finite\_viable}\ n\ s\ a\ \mathit{gt} \wedge a\ s\ i) \to (\exists\ s',\ \mathit{gt}\ s\ i\ s' \wedge \mathit{finite\_viable}\ n\ s'\ a\ \mathit{gt}).$

Lemma $\mathit{fv\_ex\_implies\_viable}$: $\forall\ n\ (a\mathord{:}$assumption$)\ \mathit{gi}\ \mathit{gt}\ (\mathit{init}\mathord{:}\mathit{initial})\ (t\mathord{:}\mathit{transition})\ (s\mathord{:}\mathit{state}),$

$(\mathit{finite\_viable}\ n\ s\ a\ \mathit{gt} \wedge \mathit{Extendcheck}\ n\ a\ \mathit{gt}) \to \mathit{viable}\ s\ a\ \mathit{gi}\ \mathit{gt}.$

Theorem $\mathit{realizable\_soundness}$: $\forall\ (s\mathord{:}\mathit{state})\ (\mathit{init}\mathord{:}\mathit{initial})\ (t\mathord{:}\mathit{transition})\ a\ \mathit{gi}\ \mathit{gt},$

$(\exists\ n, (\mathit{Basecheck}\ n\ a\ \mathit{gi}\ \mathit{gt} \wedge \mathit{Extendcheck}\ n\ a\ \mathit{gt})) \to \mathit{realizable\_contract}\ a\ \mathit{gi}\ \mathit{gt}.$

Definition $\mathit{Basecheck\_simple}$ $(n\mathord{:}\mathit{nat})$ $(a\mathord{:}$assumption$)$ $(\mathit{gi}\mathord{:}\mathit{iguarantee})$ $(\mathit{gt}\mathord{:}\mathit{tguarantee})$$:=$

$\forall\ s,\ (\mathit{gi}\ s) \to \mathit{extendable}\ n\ s\ a\ \mathit{gt}.$

Theorem $\mathit{basecheck\_soundness\_one\_way}$: $\forall\ n\ a\ (\mathit{gi}\mathord{:}\mathit{iguarantee})\ \mathit{gt}\ (i\mathord{:}\mathit{inputs}),$

$((\exists\ s,\ \mathit{gi}\ s) \wedge$

$(\forall\ k,\ (k \leq n) \to \mathit{Basecheck\_simple}\ k\ a\ \mathit{gi}\ \mathit{gt})) \to \mathit{Basecheck}\ n\ a\ \mathit{gi}\ \mathit{gt}.$

```
Definition skolem_0 (s:state) (i:inputs) : state := admit.

Definition skolem_1 (s:state) (i: inputs) (s': state) (i': inputs) : state := admit.

Inductive extendable_f : nat → state → assumption → tguarantee → Prop :=
|exnill : ∀ s (a:assumption) (gt:tguarantee), (∀ i, a s i → (gt s i (skolem_0 s i))) →
extendable_f O s a gt
|exone : ∀ s i (a:assumption) (gt:tguarantee), a s i ∧ gt s i (skolem_0 s i) ∧
   (∀ s' i', a s' i' → gt s' i' (skolem_1 s i s' i')) → extendable_f (S O) s a gt.

Theorem extf_zero_implies_ext_zero : ∀ (a:assumption) (gt:tguarantee) (s:state),
extendable_f O s a gt → extendable O s a gt.

Theorem extf_one_implies_ext_one : ∀ (a:assumption) (gt:tguarantee) (s:state),
extendable_f 1 s a gt → extendable 1 s a gt.

Definition Basecheck_simple_f (n:nat) (a:assumption) (gi:iguarantee)
(gt:tguarantee):=
∀ s, (gi s) → extendable_f n s a gt.

Definition Extendcheck_f (n:nat) (a:assumption) (gt:tguarantee) :=
∀ s a gt, extendable_f n s a gt.

Theorem BCheck_f_imp_BCheck_zero : ∀ (a:assumption) (gi:iguarantee)
(gt:tguarantee),
Basecheck_simple_f O a gi gt → Basecheck_simple O a gi gt.

Theorem BCheck_f_imp_BCheck_one : ∀ (a:assumption) (gi:iguarantee)
(gt:tguarantee),
Basecheck_simple_f 1 a gi gt → Basecheck_simple 1 a gi gt.

Theorem ECheck_f_imp_ECheck_zero : ∀ (a:assumption) (gt:tguarantee) (s:state),
Extendcheck_f O a gt → Extendcheck O a gt.

Theorem ECheck_f_imp_ECheck_one : ∀ (a:assumption) (gt:tguarantee) (s:state),
```

$Extendcheck\_f\ \ 1\ \ a\ \ gt \rightarrow Extendcheck\ \ 1\ \ a\ \ gt.$

Appendix B

Single Iteration of JSYN over Example in Figure 4.2 (Section 4.2.4)

Here we consider our example from Fig. 4.2 and demonstrate one iteration of the synthesis procedure. In particular the $\forall\exists$-formula of *ExtendCheck* is as follows:

$$\begin{aligned}
&(x_0 = 0 \lor x_0 = 1) \land \\
bias_0 &= ite(init, 0, ite(x_0 = 1, 1, -1) + bias_{-1}) \land \\
bias\_max_0 &= ite(init, false, ((bias_0 \geq 2) \lor (bias_0 \leq -2)) \lor bias\_max_{-1}) \land \\
guarantee_{10} &= ((state_0 = 0) \implies (bias_0 = 0)) \land \\
guarantee_{20} &= ite(init, true, ((state_{-1} = 0) \land x_0 = 1) \implies (state_0 = 2)) \land \\
guarantee_{30} &= ite(init, true, ((state_{-1} = 0) \land x_0 = 0) \implies (state_0 = 1)) \land \\
guarantee_{40} &= (bias\_max_0 \implies (state_0 = 3)) \land \\
guarantee_{50} &= ((state_0 = 0) \lor (state_0 = 1) \lor (state_0 = 2) \lor (state_0 = 3)) \land \\
guarantee\_all_0 &= (guarantee_{10} \land guarantee_{20} \land guarantee_{30} \land guarantee_{40} \land \\
&\quad guarantee_{50}) \land guarantee\_all_0 \land \\
bias_1 &= ite(false, 0, ite(x_0 = 1, 1, -1) + bias_0) \land \\
bias\_max_1 &= ite(false, false, ((bias_1 \geq 2) \lor (bias_1 \leq -2)) \lor bias\_max_0) \land
\end{aligned}$$

AE-VAL proceeds by constructing *MBP*-s and creating local Skolem functions. In one of the iterations, it obtains the following *MBP*:

$$(x_1 = 1 \wedge -1 = bias_0) \vee (x_1 = 0 \wedge 1 = bias_0) \wedge$$

$$\neg bias\_max_0 \wedge$$

$$(\neg(state_0 = 0)) \vee x_1 = 0 \wedge (\neg(state_0 = 0)) \vee x_1 = 1$$

and the following local Skolem function:

$$state_2 = 0 \wedge \qquad bias_2 = 0 \wedge \qquad bias\_max_2 = 0$$

In other words, the pair of the *MBP* and the local Skolem function is the synthesized implementation for some transitions of the automaton: the *MBP* specifies the source state, and the Skolem function specifies the destination state. From this example, it is clear that the synthesized transitions are from state 1 to state 0, and from state 2 to state 0. These *MBP* and the local Skolem are further encoded into the snippet of C code that after slight simplifications looks as follows:

```
...
if (((x[1] == 1 && (-1 == bias[0])) || (x[1] == 0&& (1 == bias[0])))
     && !bias_max[0]
     && (state[0] != 0 || x[1] == 0)
     && (!state[0] != 0 || x[1] == 1)) {
  bias_max[1] = 0;
  bias[1] = 0;
  state[1] = 0;
}
...
```