Validity-Guided Synthesis of Reactive Systems from Assume-Guarantee Contracts

Andreas Katis[1], Grigory Fedyukovich[2], Huajun Guo[1], Andrew Gacek[3], John Backes[3], Arie Gurfinkel[4], Michael W. Whalen[1]

[1] Department of Computer Science and Engineering, University of Minnesota
{katis001,guoxx663}@umn.edu, whalen@cs.umn.edu
[2] Department of Computer Science, Princeton University
grigoryf@cs.princeton.edu
[3] Rockwell Collins Advanced Technology Center
{andrew.gacek,john.backes}@rockwellcollins.com
[4] Department of Electrical and Computer Engineering, University of Waterloo
agurfinkel@uwaterloo.ca

Abstract. Automated synthesis of reactive systems from specifications has been a topic of research for decades. Recently, a variety of approaches have been proposed to extend synthesis of reactive systems from propositional specifications towards specifications over rich theories. We propose a novel, completely automated approach to program synthesis which reduces the problem to deciding the validity of a set of $\forall\exists$-formulas. In spirit of IC3 / PDR, our problem space is recursively refined by blocking out regions of unsafe states, aiming to discover a fixpoint that describes safe reactions. If such a fixpoint is found, we construct a witness that is directly translated into an implementation. We implemented the algorithm on top of the JKind model checker, and exercised it against contracts written using the Lustre specification language. Experimental results show how the new algorithm outperforms JKind's already existing synthesis procedure based on k-induction and addresses soundness issues in the k-inductive approach with respect to unrealizable results.

1 Introduction

Program synthesis is one of the most challenging problems in computer science. The objective is to define a process to automatically derive implementations that are guaranteed to comply with specifications expressed in the form of logic formulas. The problem has seen increased popularity in the recent years, mainly due to the capabilities of modern symbolic solvers, including Satisfiability Modulo Theories (SMT) [1] tools, to compute compact and precise regions that describe under which conditions an implementation exists for the given specification [25]. As a result, the problem has been well-studied for the area of propositional specifications (see Gulwani [15] for a survey), and approaches have been proposed to tackle challenges involving richer specifications. Template-based techniques focus on synthesizing programs that match a certain shape (the

template) [28], while *inductive synthesis* uses the idea of refining the problem space using counterexamples, to converge to a solution [12]. A different category is that of *functional synthesis*, in which the goal is to construct functions from pre-defined input/output relations [22].

Our goal is to effectively synthesize programs from safety specifications written in the Lustre [18] language. These specifications are structured in the form of *Assume-Guarantee* contracts, similarly to approaches in Linear Temporal Logic [11]. In prior work, we developed a solution to the synthesis problem which is based on k-induction [14, 19, 21]. Despite showing good results, the approach suffers from soundness problems with respect to unrealizable results; a contract could be declared as unrealizable, while an actual implementation exists. In this work, we propose a novel approach that is a direct improvement over the k-inductive method in two important aspects: performance and generality. On all models that can be synthesized by k-induction, the new algorithm always outperforms in terms of synthesis time while yielding roughly approximate code sizes and execution times for the generated code. More importantly, the new algorithm can synthesize a strictly larger set of benchmark models, and comes with an improved termination guarantee: unlike in k-induction, if the algorithm terminates with an "unrealizable" result, then there is no possible realization of the contract.

The technique has been used to synthesize contracts involving linear real and integer arithmetic (LIRA), but remains generic enough to be extended into supporting additional theories in the future, as well as to liveness properties that can be reduced to safety properties (as in k-liveness [7]). Our approach is completely automated and requires no guidance to the tools in terms of user interaction (unlike [26, 27]), and it is capable of providing solutions without requiring any templates, as in e.g., work by Beyene et. al. [2]. We were able to automatically solve problems that were "hard" and required hand-written templates specialized to the problem in [2].

The main idea of the algorithm was inspired by induction-based model checking, and in particular by IC3 / Property Directed Reachability (PDR) [4, 9]. In PDR, the goal is to discover an inductive invariant for a property, by recursively blocking generalized regions describing unsafe states. Similarly, we attempt to reach a greatest fixpoint that contains states that react to arbitrary environment behavior and lead to states within the fixpoint that comply with all guarantees. Formally, the greatest fixpoint is sufficient to prove the validity of a $\forall\exists$-formula, which states that for any state and environment input, there exists a system reaction that complies with the specification. Starting from the entire problem space, we recursively block regions of states that violate the contract, using *regions of validity* that are generated by invalid $\forall\exists$-formulas. If the refined $\forall\exists$-formula is valid, we reach a fixpoint which can effectively be used by the specified transition relation to provide safe reactions to environment inputs. We then extract a witness for the formula's satisfiability, which can be directly transformed into the language intended for the system's implementation.

The algorithm was implemented as a feature in the JKind model checker and is based on the general concept of extracting a witness that satisfies a $\forall\exists$-formula, using the AE-VAL Skolemizer [10,19]. While AE-VAL was mainly used as a tool for solving queries and extracting Skolems in our k-inductive approach, in this paper we also take advantage of its capability to generate *regions of validity* from invalid formulas to reach a fixpoint of satisfiable assignments to state variables.

The contributions of the paper are therefore:

- A novel approach to synthesis of contracts involving rich theories that is efficient, general, and completely automated (no reliance on templates or user guidance),
- an implementation of the approach in a branch of the JKind model checker, and
- an experiment over a large suite of benchmark models demonstrating the effectiveness of the approach.

The rest of the paper is organized as follows. Sect. 2 briefly describes the Cinderella-Stepmother problem that we use as an example throughout the paper. In Sect. 3, we provide the necessary formal definitions to describe the synthesis algorithm, which is presented then in Sect. 4. We present an evaluation in Sect. 5 and comparison against a method based on k-induction that exists using the same input language. Finally, we discuss the differences of our work with closely related ideas in Sect. 6 and conclude in Sect. 7.

2 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.* [3].

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 [2]. 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 paper 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.

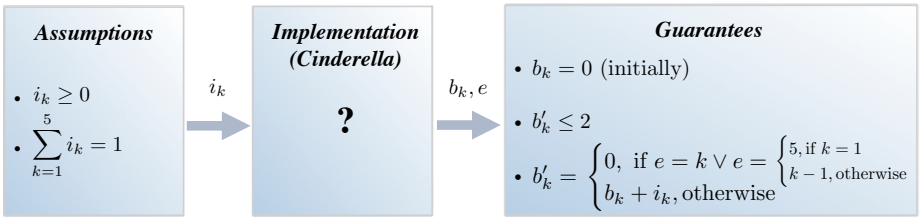


Fig. 1: An Assume-Guarantee contract.

A (conceptually) simple example is shown in Fig. 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 Fig. 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 Fig. 1) and the guarantee constraints G (right box in Fig. 1). For the particular example, it is possible to construct at least one implementation that satisfies G given A which is described in Sect. 4.3. 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 Fig. 1, we are seeking for 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 example 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.

3 Background

We use two disjoint sets, *state* and *inputs*, to describe a system. A straightforward and intuitive way to represent an *implementation* is by defining a *transition system*, composed of an initial state predicate $I(s)$ of type $\mathit{state} \rightarrow \mathit{bool}$, as well as a transition relation $T(s, i, s')$ of type $\mathit{state} \rightarrow \mathit{inputs} \rightarrow \mathit{state} \rightarrow \mathit{bool}$.

Combining the above, we represent an Assume-Guarantee (AG) contract using a set of *assumptions*, $A : \mathit{state} \rightarrow \mathit{inputs} \rightarrow \mathit{bool}$, and a set of *guarantees* G. The latter is further decomposed into two distinct subsets $G_I : \mathit{state} \rightarrow \mathit{bool}$ and $G_T : \mathit{state} \rightarrow \mathit{inputs} \rightarrow \mathit{state} \rightarrow \mathit{bool}$. The G_I defines the set of valid initial states, and G_T contains constraints that need to be satisfied in every transition

between two states. Importantly, we do not make any distinction between the internal state variables and the output variables in the formalism. This allows us to use the state variables to (in some cases) simplify the specification of guarantees since a contract might not be always defined over all variables in the transition system.

Consequently, we can formally define a realizable contract, as one for which any preceding state s can transition into a new state s' that satisfies the guarantees, assuming valid inputs. For a system to be ever-reactive, these new states s' should be further usable as preceding states in a future transition. States like s and s' are called *viable* if and only if:

$$\mathsf{Viable}(s) = \forall i.(A(s,i) \Rightarrow \exists s'.\ G_T(s,i,s') \wedge \mathsf{Viable}(s')) \tag{1}$$

This equation is recursive and we interpret it coinductively, i.e., as a greatest fixpoint. A necessary condition, finally, is that the intersection of sets of viable states and initial states is non-empty. As such, to conclude that a contract is realizable, we require that

$$\exists s.G_I(s) \wedge \mathsf{Viable}(s) \tag{2}$$

The synthesis problem is therefore to determine an initial state s_i and function $f(s,i)$ such that $G_I(s_i)$ and $\forall s, i.\mathsf{Viable}(s) \Rightarrow \mathsf{Viable}(f(s,i))$.

The intuition behind our proposed algorithm in this paper relies on the discovery of a fixpoint F that only contains viable states. We can determine whether F is a fixpoint by proving 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 both the fixpoint generation and the witness extraction, we depend on AE-VAL, a solver for $\forall\exists$-formulas.

3.1 Skolem functions and regions of validity

We rely on the already established algorithm to decide the validity of $\forall\exists$-formulas and extract Skolem functions, called AE-VAL [10]. 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 $\Phi(x,y)$ 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. We refer the reader to [19] for more details on the Skolem-function generation.

Our approach presented in this paper relies 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

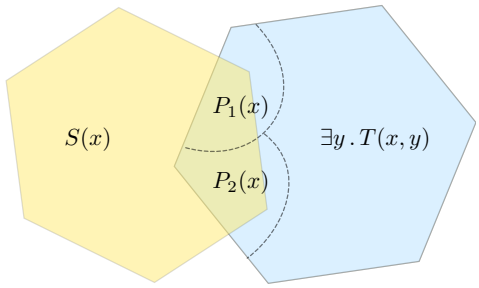


Fig. 2: Region of validity computed for an example requiring AE-VAL to iterate two times.

a common model with $S(x)$ and $P_i(x) \Rightarrow \exists y\,.\,T(x,y)$. After n iterations, AE-VAL establishes a formula $R_n(x) \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} P_i(x)$ 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. Fig. 2 shows a Venn diagram for an example of the opposite scenario: $R_2(x) = T_1(x) \vee T_2(x)$, 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*, and in this work, 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 1. *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.

4 Validity-Guided Synthesis from Assume-Guarantee Contracts

Alg. 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 Section 3 and is provided as an input. The algorithm relies on AE-VAL, for each call of which we write $\langle x, y, z\rangle \leftarrow$ AE-VAL$(\ldots)$: x specifies if the given formula is *valid* or *invalid*, y identifies the region of validity (in both cases), and z – the Skolem function (only in case of the validity).

Algorithm 1 JSyn-vg (A : assumptions, G : guarantees)

1: $F(s) \leftarrow \mathit{true}$; ▷ Fixpoint of viable states
2: **while** *true* **do**
3: $\phi \leftarrow \forall s, i.\ (F(s) \wedge A(s,i) \Rightarrow \exists s'.G_T(s,i,s') \wedge F(s'))$;
4: $\langle \mathit{valid}, \mathit{validRegion}, \mathit{Skolem} \rangle \leftarrow$ AE-VAL(ϕ);
5: **if** *valid* **then**
6: **if** $\exists s.G_I(s) \wedge F(s)$ **then**
7: **return** $\langle$REALIZABLE, *Skolem*, $s, F\rangle$;
8: **else** ▷ Empty set of initial or viable states
9: **return** UNREALIZABLE;
10: **else** ▷ Extract region of validity $Q(s,i)$
11: $Q(s,i) \leftarrow \mathit{validRegion}$;
12: $\phi' \leftarrow \forall s.\ (F(s) \Rightarrow \exists i.A(s,i) \wedge \neg Q(s,i))$;
13: $\langle \_, \mathit{violatingRegion}, \_ \rangle \leftarrow$ AE-VAL(ϕ');
14: $W(s) \leftarrow \mathit{violatingRegion}$;
15: $F(s) \leftarrow F(s) \wedge \neg W(s)$; ▷ Refine set of viable states

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 Eqs. 1 and 2), i.e., a greatest fixpoint is reached. We first encode Eq. 1 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 a safe reaction, 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 formula in Eq. 2 by a simple satisfiability check. 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 7). 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 15, 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 9).

4.1 Soundness

Lemma 2. $\mathsf{Viable} \Rightarrow F$ *is an invariant for Alg. 1.*

Proof. It suffices to show this invariant holds each time F is assigned. On line 1, this is trivial. For line 15, we can assume that $\mathsf{Viable} \Rightarrow F$ holds prior to this line. Suppose towards contradiction that the assignment on line 15 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 1. 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 1. 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 15, 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 15.

Theorem 1. *The* REALIZABLE *and* UNREALIZABLE *results of Alg. 1 are sound.*

Proof. If Alg. 1 terminates, then the formula for ϕ on line 3 is valid. Rewritten, F satisfies the formula

$$\forall s.\ F(s) \Rightarrow (\forall i.\ A(s,i) \Rightarrow \exists s'.G_T(s,i,s') \wedge F(s'))\,. \tag{3}$$

Let the function f be defined over state predicates as

$$f = \lambda V.\lambda s.\ \forall i.\ A(s,i) \Rightarrow \exists s'.G_T(s,i,s') \wedge V(s'). \tag{4}$$

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 Eq. 3 we have $F \Rightarrow f(F)$. Thus F is a post-fixed point of f. In Eq. 1, Viable is defined as the greatest fixed-point of f. Thus $f \Rightarrow \mathsf{Viable}$ by the Knaster-Tarski theorem. Combining this with Lemma 2, we have $F = \mathsf{Viable}$. Therefore the check on line 7 is equivalent to the check in Eq. 2 for realizability.

4.2 Termination on finite models

Lemma 3. *Every loop iteration in Alg. 1 either terminates or removes at least one state from F.*

```
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;
```

Fig. 3: An Assume-Guarantee contract for the Cinderella-Stepmother game in Lustre.

Proof. It suffices to show that at least one state is removed from F on line 15. That is, we want to show that $F \cap W \neq \varnothing$ since this intersection is what is removed from F by line 15.

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 1. 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 15 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 15.

Theorem 2. *For finite models, Alg. 1 terminates.*

Proof. Immediately from Lemma 3 and the fact that AE-VAL terminates on finite models [10].

4.3 Applying JSyn-vg to the Cinderella-Stepmother game

Fig. 3 shows one possible interpretation of the contract designed for the instance of the Cinderella-Stepmother game that we introduced in Sect. 2. The contract is expressed in Lustre [18], a language that has been extensively used for specification as well as implementation of safety-critical systems, and is the kernel language in SCADE, a popular tool in model-based development. The contract is defined as a Lustre 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: $A(i_1, \ldots, i_5) = (\bigwedge_{k=1}^{5} i_k >= 0.0) \wedge (\sum_{k=1}^{5} i_k = 1.0)$. 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 specific bucket, or an adjacent one. If so, the value of each $\mathtt{b}_k$ at the the next turn becomes equal to the value of the corresponding input variable $\mathtt{i}_k$. 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 adding Stepmother's distribution 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 Fig. 3 as input to Alg. 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 $\forall i_1, \ldots, i_5, b_1, \ldots, b_5 \,.\, A(i_1, \ldots, i_5) \Rightarrow \exists b'_1, \ldots, b'_5, e \,.\, G_T(i_1, \ldots, i_5, b_1, \ldots, b_5, b'_1, \ldots, b'_5, e)$. 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 $cex = \{\ldots, b_4 = 3025, i_4 = 0.2, b'_4 = 3025.2, \ldots\}$, 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 in the paper; intuitively it contains some con-

straints 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 $\forall b_1, \ldots, b_5 \,.\, \exists i_1, \ldots, i_5 \,.\, A(i_1, \ldots, i_5) \wedge \neg Q(i_1, \ldots, i_5, b_1, \ldots, b_5)$ and extracts a violating region W over $b_1, \ldots, b_5$. Precise encoding of W is also too large to be presented in the paper; and 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 Implementation and Evaluation

The implementation of the algorithm has been added to a branch of the JKind [13] model checker[5]. JKind officially supports synthesis using a k-inductive approach, named JSyn [19]. For clarity, we named our validity-guided technique JSyn-vg (i.e., validity-guided synthesis). JKind uses Lustre [18] 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[6]. Skolem functions returned by AE-VAL get then translated into an efficient and practical implementation. To compare the quality of implementations against JSyn, we use SMTLib2C, a tool that has been specifically developed to translate Skolem functions to C implementations[7].

5.1 Experimental results

We evaluated JSyn-vg by synthesizing implementations for 124 contracts[8] originated 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

[5] The JKind fork with JSyn-vg is available at `https://goo.gl/WxupTe`.
[6] The AE-VAL tool is available at `https://goo.gl/CbNMVN`.
[7] The SMTLib2C tool is available at `https://goo.gl/EvNrAU`.
[8] All of the benchmark contracts can be found at `https://goo.gl/2p4sT9`.

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 [16];
- 11 models 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 [2] (this was the only synthesis problem in the paper). We examined the following aspects:

- 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 1. Fig. 4a shows the time allocated 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%. Fig. 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 entire space of states is an inductive invariant. However, several spikes in Fig. 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. Interetsingly, 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.

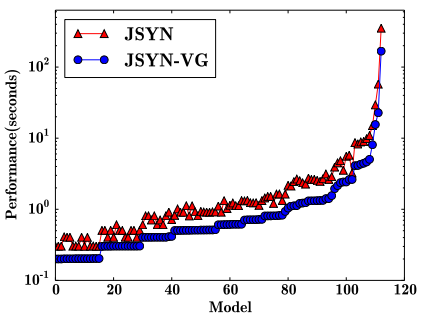
One last statistic that we tracked was the performance of the synthesized C implementations in terms of execution time, which can be seen in Fig. 4c. The performance was computed as the mean of 1000000 iterations of executing each

Table 1: Benchmark statistics.

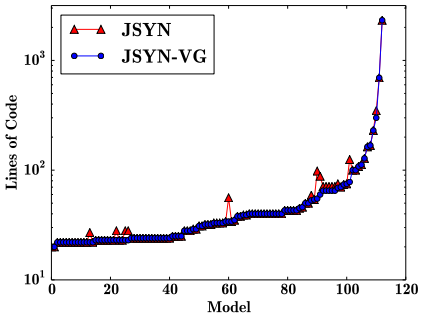
	JSyn	JSyn-vg
Problems solved	113	**124**
Performance (avg - seconds)	5.72	**2.78**
Performance (max - seconds)	352.1	**167.55**
Implementation Size (avg - Lines of Code)	72.88	**70.66**
Implementation Size (max - Lines of Code)	2322	**2142**
Implementation Performance (avg - ms)	57.84	**56.32**
Implementation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

Table 2: Cinderella-Stepmother results.

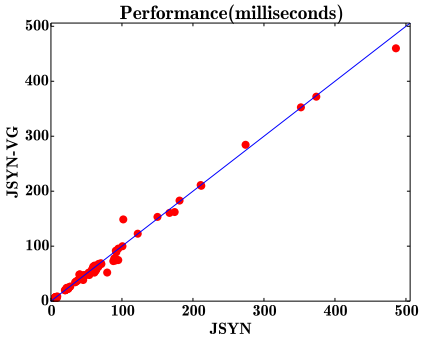
Game	JSyn-vg			ConSynth [2]	
	Impl. Size (LoC)	Impl. Performance (ms)	Time	Time (Z3)	Time (Barcelogic)
Cind (C = 3)	204	128.09	4.5s	3.2s	1.2s
Cind2 (C = 3)	2081	160.87	28.7s		
Cind (C = 2)	202	133.04	4.7s	1m52s	1m52s
Cind2 (C = 2)	1873	182.19	27.2s		



(a) Performance of synthesizers



(b) Size of implementations



(c) Performance of implementations

Fig. 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
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Fig. 5: Spurious counterexample for Cinderella-Stepmother example using JSyn

implementation using random input values. According to the figure as well as Table 1, the differences are minuscule on average.

Fig. 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 Fig. 3 as an input to JSyn, we receive an "unrealizable" answer, with the counterexample shown in Fig. 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 2 shows how JSyn-vg performed on the four contracts describing the Cinderella-Stepmother game. We used two different interpretations for the game, and exercised both for the cases where the bucket capacity `C` is equal to 2 and 3. Regarding the synthesized implementations, their size is analogous to the complexity of the program (Cinderella2 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 [2]. 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 not tied to a specific environment since it can be extended to support more theories, as well as categories of specification.

6 Related Work

The work presented in this paper is closely related to approaches that attempt to construct infinite-state implementations. Some focus on the continuous interaction of the user with the underlying machinery, either through the use of templates [2, 28], or environments where the user attempts to guide the solver by choosing reactions from a collection of different interpretations [26]. In contrast, our approach is completely automatic and does not require human ingenuity to find a solution. Most importantly, the user does not need to be deeply familiar with the problem at hand.

Iterative strengthening of candidate formulas is also used in abductive inference [8] 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 [8] for reactive synthesis.

The concept of synthesizing implementations by discovering fixpoints was mostly inspired by the IC3 / PDR [4, 9], 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 [5, 6]. Discovering fixpoints to synthesize reactive designs was first extensively covered by Piterman *et al.* [23] 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 and avoids the partitioning of the input space. As the tools use different notations and support different logical fragments, practical comparisons are not straightforward, and thus are left for the future.

More recently, Preiner *et al.* presented work on model synthesis [24], that employs a counterexample-guided refinement process [25] 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. In general, enumeration techniques such as the one used in ConSynth's underlying E-HSF engine [2] 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.

7 Conclusion and Future Work

We 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. 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. In the future, we would like to extend the applicability of this algorithm to other areas in formal verification, such as invariant generation. Another interesting goal is to make the proposed benchmark collection available to competitions such as SYNTCOMP, by establishing a formal extension for the TLSF format to support infinite-state problems [17]. Finally, a particularly interesting challenge is that of mapping infinite theories to finite counterparts, enabling the synthesis of secure and safe implementations.

8 Data Availability Statement

The datasets generated during and/or analyzed during the current study are available in the figshare repository: `https://doi.org/10.6084/m9.figshare.5904904` [20].

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
2. Beyene, T., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: POPL. pp. 221–233. ACM (2014)
3. Bodlaender, M.H.L., Hurkens, C.A., Kusters, V.J., Staals, F., Woeginger, G.J., Zantema, H.: Cinderella versus the wicked stepmother. In: IFIP TCS. LNCS, vol. 7604, pp. 57–71. Springer (2012)
4. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: VMCAI. LNCS, vol. 6538, pp. 70–87. Springer (2011)
5. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Parameter synthesis with IC3. In: FMCAD. pp. 165–168. IEEE (2013)
6. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: TACAS. LNCS, vol. 9035, pp. 52–67. Springer (2015)
7. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: Formal Methods in Computer-Aided Design (FMCAD), 2012. pp. 52–59. IEEE (2012)
8. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. In: OOPSLA. pp. 443–456. ACM (2013)
9. Een, N., Mishchenko, A., Brayton, R.: Efficient Implementation of Property Directed Reachability. In: FMCAD. pp. 125–134. IEEE (2011)
10. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated Discovery of Simulation Between Programs. In: LPAR. LNCS, vol. 9450, pp. 606–621. Springer (2015)
11. Firman, E., Maoz, S., Ringert, J.O.: Performance Heuristics for GR(1) Synthesis and Related Algorithms. In: SYNT@CAV. EPTCS, vol. 260, pp. 62–80. Open Publishing Association (2017)
12. Flener, P., Partridge, D.: Inductive programming. Autom. Softw. Eng. 8(2), 131–137 (2001)
13. Gacek, A.: JKind – an infinite-state model checker for safety properties in Lustre. http://loonwerks.com/tools/jkind.html (2016)
14. Gacek, A., Katis, A., Whalen, M.W., Backes, J., Cofer, D.: Towards Realizability Checking of Contracts Using Theories. In: NFM. LNCS, vol. 9058, pp. 173–187. Springer (2015)
15. Gulwani, S.: Dimensions in program synthesis. In: PPDP. pp. 13–24. ACM (2010)
16. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: FMCAD. pp. 1–9. IEEE (2008)
17. Jacobs, S., Klein, F., Schirmer, S.: A high-level LTL synthesis format: TLSF v1.1. In: Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016. pp. 112–132 (2016), https://doi.org/10.4204/EPTCS.229.10
18. Jahier, E., Raymond, P., Halbwachs, N.: The Lustre V6 Reference Manual, http://www-verimag.imag.fr/Lustre-V6.html
19. Katis, A., Fedyukovich, G., Gacek, A., Backes, J.D., Gurfinkel, A., Whalen, M.W.: Synthesis from Assume-Guarantee Contracts using Skolemized Proofs of Realizability. CoRR abs/1610.05867 (2016), http://arxiv.org/abs/1610.05867
20. Katis, A., Fedyukovich, G., Guo, H., Gacek, A., Backes, J., Gurfinkel, A., Whalen, M.W.: Validity-guided synthesis of reactive systems from assume-guarantee contracts. Figshare (2018), https://doi.org/10.6084/m9.figshare.5904904

21. Katis, A., Gacek, A., Whalen, M.W.: Towards synthesis from assume-guarantee contracts involving infinite theories: a preliminary report. In: FormaliSE. pp. 36–41. IEEE (2016)
22. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT 15(5-6), 455–474 (2013)
23. Piterman, N., Pnueli, A., SaŠar, Y.: Synthesis of Reactive(1) Designs. In: VMCAI. LNCS, vol. 3855, pp. 364–380. Springer (2006)
24. Preiner, M., Niemetz, A., Biere, A.: Counterexample-guided model synthesis. In: TACAS. pp. 264–280. Springer (2017)
25. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: CAV, Part II. LNCS, vol. 9207, pp. 198–216. Springer (2015)
26. Ryzhyk, L., Walker, A.: Developing a practical reactive synthesis tool: Experience and lessons learned. arXiv preprint arXiv:1611.07624 (2016)
27. Ryzhyk, L., Walker, A., Keys, J., Legg, A., Raghunath, A., Stumm, M., Vij, M.: User-guided device driver synthesis. In: OSDI. pp. 661–676 (2014)
28. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. STTT 15(5-6), 497–518 (2013)