Synthesis from Assume-Guarantee Contracts using Skolemized Proofs of Realizability

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

[1] Department of Computer Science and Engineering, University of Minnesota, 200 Union Street, Minneapolis, MN 55455, USA
katis001@umn.edu, whalen@cs.umn.edu
[2] Computer Science and Engineering, University of Washington, Seattle, WA, USA
grigory@cs.washington.edu
[3] Rockwell Collins Advanced Technology Center 400 Collins Rd. NE, Cedar Rapids, IA, 52498, USA
{andrew.gacek,john.backes}@rockwellcollins.com
[4] Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
agurfinkel@uwaterloo.ca

Abstract. The realizability problem in requirements engineering is to decide existence of an implementation that meets the given formal requirements. A step forward after the realizability is proven is to construct such an implementation automatically, and thus solve the problem of program synthesis. In this paper, we propose a novel approach to program synthesis guided by the proofs of realizability represented by the set of valid $\forall\exists$-formulas. In particular, we propose to extract Skolem functions witnessing the existential quantification, and to compose the Skolem functions into an implementation that is guaranteed to comply with the user-defined requirements. We implemented the approach for requirements in the form of Assume-Guarantee contracts, using the Lustre specification language. It naturally extends the realizability check by the JKind model checker. Furthermore, we developed a compiler to translate pure Skolem-containing implementations to the C programming language. For a vast variety of models, we test their corresponding implementations against the ones provided by the LustreV6 compiler, yielding meaningful results.

Mike: What should tool name be? KindSyn, RealSyn, jsyn "jason". I have defined it by command to make it easy to replace.

1 Introduction

Automated synthesis research is concerned with discovering efficient algorithms to construct candidate programs that are guaranteed to comply with predefined temporal specifications. This problem has been well studied for propositional specifications, especially for (subsets of) LTL [25]. More recently, the

problem of synthesizing programs for richer theories has been examined, including work in *template synthesis* [47], which attempts to find programs that match a certain shape (the template), and *inductive synthesis*, which attempts to use counterexample-based refinement to solve synthesis problems [20]. Such techniques have been widely used for stateless formulas over arithmetic domains [45]. *Functional synthesis* has also been effectively used to synthesize subcomponents of already existing partial implementations [38,39]. In this paper, we propose a new approach that can synthesize programs for arbitrary *assume/guarantee contracts* that do not have to conform to specific template shapes or temporal restrictions. The contracts are described using safety properties involving real arithmetic. Although the technique is not guaranteed to succeed or terminate, we have used it to successfully synthesize a range of programs over non-trivial contracts. The power of expressiveness that is provided by the Assume-Guarantee framework acts as the main catalyst behind the vast variety of applications that can be supported by our synthesis procedure, involving instances from all of the different research areas that were mentioned in the beginning of this Section. As such, it can be used both with template-based specifications, support temporal safety properties, potentially support counterexample refinement as an intermediate optimization, as well as construct program fragments that correspond to specification for leaf-level system components.

Our approach is built on previous work determining the *realizability* of contracts involving infinite theories such as linear integer/real arithmetic and/or uninterpreted functions [22,33]. The algorithm, explained in Section 2, uses a quantified variant of k-induction that can be checked by any SMT solver that supports quantification. Notionally, it checks whether a sequence of states satisfy the contract of depth k is sufficient to guarantee the existence of a successor state that satisfies the contract for an arbitrary input. An outer loop of the algorithm increases k until either a solution or counterexample is found.

The step from realizability to synthesis involves moving from the existence of a witness (as can be provided by an SMT solver such as z3 or cvc4) to the witness itself. For this, the most important obstacle is the (in)ability of the SMT solver to handle higher-order quantification. Fortunately, interesting directions to solving this problem have already surfaced, either by extending an SMT solver with native synthesis capabilities[45], or by providing external algorithms that reduce the problem by efficient quantifier elimination methods [18]. Our synthesis relies on our previous implementation for realizability checking and the skolemization procedure implemented in the AE-VAL tool [18].

We combined the above ideas to create a reasonable sequential synthesis approach, which we call RealSynth. It applies the realizability checker from [22] and then extracts a Skolem witness formula from the AE-VAL tool that can immediately be turned into a C program. In order to support synthesis, several changes were required to the quantifier-elimination approach to produce Skolem *functions* rather than *relations*. The main contributions of the work are therefore:

- To the best of our knowledge, the first synthesis procedure from Assume-Guarantee contracts, modulo infinite theories, usable in a broader list of applications when compared to already existing approaches.
- Mike: GRIGORY: Something about improvements to AEVal to support synthesis
- A prototype tool implementing the algorithm
- An experiment demonstrating the application of the tool on various benchmark examples

This paper presents the first full exposition of the idea, which was originally proposed in a workshop paper without an implementation or experiment [34].

In Section 2 we provide the necessary background definitions that are used in our synthesis algorithm, as well as an informal proof of the algorithm's correctness. Section 3 contains the core formal notions behind on which the AE-VAL Skolemizer is based, as well as the adjustments that were done for it to better support the needs of this work. Section 4 provides detailed source information for each one of the important components of this work. Section 5 presents our results on using the algorithm to automatically generate leaf-level component implementations for different case studies. Finally, in Section 6 we give a brief historical background on the related research work on synthesis, and we conclude with a discussion on potential future work in Section 7.

2 Synthesis from Assume-Guarantee Contracts

In this section we provide a brief background on Assume-Guarantee contracts, proceed with summarizing our earlier results on realizability checking of contracts, and finally present our program synthesis procedure.

2.1 Assume-Guarantee Contracts

In the context of requirements engineering, there have been a lot of proposed ideas in terms of how requirements can be represented and expressed during system design. Grigory: need for citations with examples of "a lot" of ideas? One of the most popular ways to describe these requirements is through the notion of an Assume-Guarantee contract, where the requirements are expressed using safety properties that are split into two separate categories. The *assumptions* of the contract correspond to properties that restrict the set of valid inputs a system can process, while the *guarantees* dictate how the system should behave, using properties that precisely describe the kinds of valid outputs that it may return to its environment.

As an illustrative example, consider the contract specified in Figure 2. The component to be designed consists of two inputs, x and y and one output z. If we restrict our example to the case of integer arithmetic, we can see that the contract assumes that the inputs will never have the same value, and requires that the output of the component is Boolean whose value depends on the comparison of

Fig. 1: Example of an Assume-Guarantee contract

the values of x and y. Also, notice that in the middle of the figure we depict the component using a question mark symbol. The question mark simply expresses the fact that during the early stages of software development, the implementation is absent or exists only partially.

Deciding existence of an implementation for the question-mark component that satisfies the specific contract for all possible inputs is aimed by the problem of *realizability*, while automatically constructing a witness of the proof of realizability of the contract is aimed by problem of *program synthesis*. The contract in Figure 2 is obviously *realizable*, and therefore an implementation of the question-mark component exists. Interestingly, if the assumption would be omitted then the contract is clearly *unrealizable*, since no implementation is able to provide a correct output in the case where $x = y$.

2.2 Formal Preliminaries

For the purposes of this paper, we are describing a system using the types *state* and *inputs*. Formally, an *implementation*, i.e. a *transition system* can be described using a set of initial states $I(s)$ of type $state \implies bool$, in addition to a transition relation $T(s, i, s')$ that implements the contract and has type $state \implies inputs \implies state \implies bool$.

An Assume-Guarantee contract can be formally defined by two sets, a set of *assumptions* and a set of *guarantees*. The *assumptions* A impose constraints over the inputs, while the *guarantees* G are used for the corresponding constraints over the outputs of the system and can be expressed as two separate subsets G_I and G_T, where G_I defines the set of valid initial states, and G_T specifies the properties that need to be met during each new transition between two states. Note that we do not necessarily expect that a contract would be defined over all variables in the transition system, 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.

2.3 Realizability of Contracts

The synthesis algorithm proposed in this paper is built on top of our realizability algorithm originally presented in [22]. Using the formal foundations described in Sect. 2.2, the problem of realizability is expressed using the notion of a state being *extendable*:

Definition 1 (One-step extension). *A state s is extendable after n steps, denoted $Extend_n(s)$, if any valid path of length $n-1$ starting from s can be extended in response to any input.*

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

The algorithm for realizability uses Def. 1 in two separate checks that correspond to the two traditional cases exercised in k-induction. For the *BaseCheck*, we ensure that all initial states are extendable in terms of any path of length $k \leq n$, while the inductive step of *ExtendCheck* tries to prove that all valid states are extendable. Therefore, we attempt to find the smallest n, for which the two following $\forall\exists$-formulas are valid:

$$BaseCheck(n) \triangleq \forall k \leq n.(\forall s.G_I(s) \implies Extend_k(s)) \tag{1}$$

$$ExtendCheck(n) \triangleq \forall s.Extend_n(s) \tag{2}$$

The realizability checking algorithm has been used to effectively find cases where the traditional consistency check (i.e. the existence of an assignment to the input variables for which the output variables satisfy the contract) failed to detect conflicts between stated requirements in case studies of different complexity and importance. It has also been formally verified using the Coq proof assistant in terms of its soundness, for the cases where it reports that a contract is realizable [33].

2.4 Program Synthesis from Proofs of Realizability

The most important outcome of our previous work on realizability is that it can be further used for solving the more complex problem of *program synthesis* i.e., to automatically derive implementations, from the proof of the contract's realizability.

The idea behind our approach to solving the synthesis problem is simple and elegant. Consider checks (1) and (2) that are used in the realizability checking algorithm. Both checks require that the reachable states explored are extendable using Def. 1. The key insight then is to decide if $Extend_n(s)$ is valid and generate a witness for each of the n times that we run *BaseCheck* and a final witness for the inductive case in *ExtendCheck*.

In the first order logic, witnesses for valid $\forall\exists$-formulas are represented by the Skolem functions. Intuitively, a Skolem function expresses a connection between all universally quantified variables in the left-hand-side of the $\forall\exists$-formulas (1) and (2) and the existentially quantified variable s' in the right-hand-side of the the formulas. Our algorithm automatically generates such Skolem functions while solving the validity of (1) and (2) and is described in details Sect. 3.

Algorithm 1: Synthesis from Assume-Guarantee Contracts

Input: Assume-Guarantee Contract in Lustre, (A, G)
Output: Skolem collection *Skolems*,
Return value $\in$ {REALIZABLE, UNREALIZABLE} of (A, G).

```
1  i ← 0;
2  BaseResult ← BaseCheckEngine.AE-VAL(BaseCheck(i));
3  ExtendResult ← ExtendCheckEngine.AE-VAL(ExtendCheck(i));
4  forever do
5  |  if (BaseCheckEngine.isValid (BaseResult)) then
6  |  |  return UNREALIZABLE
7  |  Skolems.Add(BaseResult.Skolem);
8  |  if (ExtendCheckEngine.isInvalid(ExtendResult)) then
9  |  |  Skolems.Add((ExtendResult.Skolem));
10 |  |  return Skolems, REALIZABLE
11 |  i++;
```

Algorithm 2: Structure of implementation.

```
1  ASSIGN_GI_WITNESS_TO_S(); ▷ Initialize state values in arrays of size k each.
2  READ_INPUTS();                          ▷ Transition using BaseCheck witness
3  SKOLEMS[0]();
4  ...
5  READ_INPUTS();
6  SKOLEMS[K-1]();
7  forever do
8  |  READ_INPUTS();                       ▷ Transition using ExtendCheck witness
9  |  SKOLEMS[K]();
10 |  UPDATE_ARRAY_HISTORY();
```

Algorithm 1 provides a summary of the synthesis procedure, showing how it naturally extends our previous work on realizability checking. During the k-induction algorithm, two parallel engines (BASECHECKENGINE, EXTENDCHECKENGINE) correspondingly handle the base and inductive step checks of validity of $\forall\exists$-formulas. The proof of a formula's validity is closely tied to the process of Skolemization. As a result, every step for which the *BaseCheck(n)* is valid according to AE-VAL, we also receive a Skolem function that can be used as a witness that satisfies the formula. We keep repeating this process, accumulating Skolem functions as long as the corresponding *BaseCheck(n)* is valid. As soon as the inductive step of *ExtendCheck(n)* 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 collection of the Skolem functions to the user.

Given a collection of Skolems, it remains to plug them into an implementation skeleton of shown in Alg. 2. One of the direct effects of using Lustre in

conjunction with a k-inductive proof is that we can have properties in the model that refer to up to a previous value of a variable, up to $k-1$ steps in the past. As such, the implementation begins (method ASSIGN_GI_WITNESS_TO_S()) by creating an array for each state variable up to depth k, where k is the depth at which we found a solution to our realizability algorithm. In each array, the i-th element, with $0 \leq i \leq k$, corresponds to the value assigned to the variable after the call to i-th Skolem function. As such, the k elements of the array correspond to the k Skolem functions produced by the *BaseCheck* process, while the last element is also used by the Skolem function generated from the formula corresponding to the *ExtendCheck* process.

The algorithm then uses the Skolem functions generated by AE-VAL for each of the *BaseCheck* instances to describe the initial behavior of the implementation up to depth k. This process starts from the memory-free description of the initial state (G_I). There are two "helper" operations: UPDATE_ARRAY_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 input variable arrays, where i represents the i-th step of the process. Once the history is entirely initialized using the *BaseCheck* witness values, we add the Skolem function that represents the witness for the *ExtendCheck* instance to describe the recurrent behavior of the implementation, i.e., the next value of outputs in each iteration in the infinite loop.

Finally, to further strengthen our claims regarding the algorithm's correctness, we wrote machine-checked proofs regarding the validity of *BaseCheck(n)* and *ExtendCheck(n)*, when Skolem functions are used as witness states towards synthesizing the implementations. The entirety of the models explored in this paper only involved proofs of realizability of length k equal to 0 or 1[5]. As such, we limited our proofs of soundness to these two specific cases. We hope to extend the proofs to capture any arbitrary k as part of our future work. The corresponding were written and proved using the Coq proof assistant [48].

Theorem 1 (Bounded Soundness of BaseCheck and ExtendCheck using Skolem Functions). *Let* $BaseCheck_{Skolem_n}(n)$ *and* $ExtendCheck_{Skolem_n}(n)$, $n \in 0, 1$ *be the valid variations of the corresponding formulas BaseCheck(n) and ExtendCheck(n), where the existentially quantified part has been substituted with a witnessing Skolem function. We have that:*

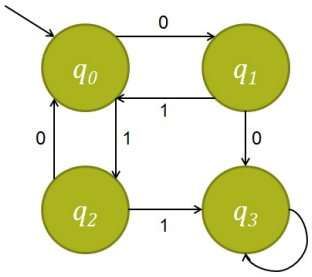
- $\forall(A, G_I, GT).BaseCheck(n) \Rightarrow BaseCheck_{Skolem_n}(n)$
- $\forall(A, G_I, GT).ExtendCheck(n) \Rightarrow ExtendCheck_{Skolem_n}(n)$

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. □

2.5 Running Example

Fig. 2 shows a simple but yet representative example to our approach. This is the set of requirements in the Lustre language and an automaton satisfying

[5] The proofs can be found at `https://github.com/andrewkatis/Coq`.




```
node top(x : bool; state: int) returns (  );
var
  bias : int;
  prop1, prop2, prop3, prop4, prop_all : bool;
  bias_max : bool;
let
  bias = 0 -> (if x then 1 else -1) + pre(bias);
  bias_max = false ->
        (bias >= 2 or bias <= -2) or pre(bias_max);
  prop1 = (state = 0 => (bias = 0));
  prop2 = true ->
          (pre(state = 0) and x) => state = 2;
  prop3 = true ->
          (pre(state = 0) and not x) => state = 1;
  prop4 = bias_max => state = 3;
  prop5 = state = 0 or state = 1 or state = 2 or state = 3;
  prop_all = prop1 and prop2 and prop3 and prop4 and prop5;
  --%PROPERTY prop_all;
  --%REALIZABLE x;
tel;
```

Fig. 2: Automaton and Requirements for running example

these requirements. There is an input variable `x` (used as an argument to the `-%REALIZABLE` query) and an output variable `state`. There are five properties. Properties `prop2` and `prop3` are used to indirectly describe some possible transitions in the automaton. Properties `prop1` and `prop4` are the requirements with respect to two local variables, `bias` and `bias_max`. Variable `bias` calculates the number of successive ones or zeros read by the automaton, while `bias_max` is used as a flag to indicate that at least two zeros or two ones have been read in a row. Finally, property `prop5` specifies the range of values of variable `state`.

Considering these properties, we want to give an answer to whether an implementation can be constructed, that is guaranteed to satisfy the constraints imposed by the specification. In other words, we want to prove that the specification is realizable (–%REALIZABLE query). It is straightforward to conclude that the model from the example realizable, and has a k-inductive proof of length $k = 1$. 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 A for the certain formulas).

The Skolem functions are used to construct the final implementation following the outline provided in Alg. 2. It is 135 lines of code, and due to the space constraints we do not present it here[6]. 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.

[6] The implementation for the example is available at `https://arxiv.org/abs/1610.05867`.

3 Witnessing existential quantifiers with AE-VAL

Quantifier elimination is a decision procedure that turns a quantified formula into an equivalent quantifier-free formula. In addition, the quantifier elimination algorithms are often able to discover a Skolem function that represents witnesses for the existentially quantified individual variables (e.g., [2,**?**,30,32]). Various tasks in verification and synthesis [10,4,23] rely on efficient techniques to remove existential quantifiers from formulas in first order logic, thus adjusting the task to be decided by an SMT solver. In particular, *functional synthesis* aims at computing a function that meets a given input/output relation. A function with an input x and an output y, specified by a relation $f(x, y)$, can be constructed as a by-product of deciding validity of the formula $\forall x \exists y \,.\, f(x, y)$. Due to a well-known *AE-paradigm* (also referred to as *Skolem paradigm* [44]), the formula $\forall x \exists y \,.\, f(x, y)$ is equivalent to the formula $\exists sk \; \forall x \,.\, f(x, sk(x))$, which means existence of a Skolem function sk, such that $f(x, sk(x))$ holds for every x. Thus the key feature in modern quantifier elimination approaches is their ability to produce witnessing Skolem function.

In the rest of the section, we briefly describe the prior work on AE-VAL to be able (in Sect. 3.3) to present the key contributions on delivering Skolem functions appropriate for the program synthesis from proofs of realizability.

3.1 Model-Based Projection for Linear Rational Arithmetic

Quantifier elimination of a formula $\exists \vec{y} \,.\, T(\vec{x}, \vec{y})$ is an expensive procedure that typically proceeds by enumerating all models of an extended formula $T(\vec{x}, \vec{y})$. However, in some applications, the quantifier-free formula, fully equivalent to $\exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, is not even needed. Instead, it is enough to operate by (possibly incomplete) sets of models. This idea relies on some notion of projection that under-approximates existential quantification. In this section, we consider a concept of Model-Based Projections (MBP), recently proposed by [37,15].

Definition 2. *An $MBP_{\vec{y}}$ is a function from models of $T(\vec{x}, \vec{y})$ to $\vec{y}$-free formulas iff:*

$$\text{if } m \models T(\vec{x}, \vec{y}) \text{ then } m \models MBP_{\vec{y}}(m, T) \tag{3}$$

$$MBP_{\vec{y}}(m, T) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y}) \tag{4}$$

There are finitely many MBPs for fixed $\vec{y}$ and T and different models $m_1, \ldots, m_n$ (for some n): $T_1(\vec{x}), \ldots, T_n(\vec{x})$, such that $\exists \vec{y} \,.\, T(\vec{x}, \vec{y}) = \bigvee_{i=1}^{n} T_i(\vec{x})$.

A possible way of implementing an MBP-algorithm was proposed in [37]. It is based on Loos-Weispfenning (LW) quantifier-elimination method [40] for Linear Rational Arithmetic (LRA). Consider formula $\exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, where T is quantifier-free. In our simplified presentation, $\vec{y}$ is singleton, T is in Negation Normal Form (that allows the operator $\neg$ to be applied only to variables), and y appears in the literals only of the form $y = e$, $l < y$ or $y < u$, where l, u, e are y-free. LW states that the equation (5) holds:

Algorithm 3: AE-VAL$\Big(S(\vec{x}), \exists \vec{y} \,.\, T(\vec{x}, \vec{y})\Big)$, cf. [19]

Input: $S(\vec{x}), \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$.
Output: Return value $\in \{\text{VALID}, \text{INVALID}\}$ of $S(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$.
Data: SmtSolver, counter i, models $\{m_i\}$, MBPs $\{T_i(\vec{x})\}$, conditions $\{\phi_i(\vec{x}, \vec{y})\}$.

```
1  SmtAdd(S(x⃗));
2  i ← 0;
3  forever do
4  |  i++;
5  |  if (isUnsat(SmtSolve())) then return VALID;
6  |  SmtPush();
7  |  SmtAdd(T(x⃗, y⃗));
8  |  if (isUnsat(SmtSolve())) then return INVALID;
9  |  m_i ← SmtGetModel();
10 |  (T_i, φ_i(x⃗, y⃗)) ← GetMBP(y⃗, m_i, T(x⃗, y⃗)));
11 |  SmtPop();
12 |  SmtAdd(¬T_i);
```

$$\exists y \,.\, T(\vec{x}) \equiv \Big(\bigvee_{(y=e) \in lits(T)} T[e] \vee \bigvee_{(l<y) \in lits(T)} T[l+\epsilon] \vee T[-\infty] \Big) \tag{5}$$

In (5), $lits(T)$ denote the set of literals of T, $T[\cdot]$ stands for a *virtual substitution* for the literals containing y. In particular, $T[e]$ substitutes exact values of y ($y = e$), $T[l + \epsilon]$ substitutes the intervals ($l < y$) of possible values of y, $T[-\infty]$ substitutes the rest of the literals. Consequently, a function $LRAProj_T$ is an implementation of the *MBP* function for (5):

$$LRAProj_T(m) = \begin{cases} T[e], & \text{if } (y = e) \in lits(T) \wedge m \models (y = e) \\ T[l+\epsilon], & \text{else if } (l < y) \in lits(T) \wedge m \models (l < y) \wedge \\ & \quad \forall (l' < y) \in lits(T) \,.\, m \models \big((l' < y) \implies (l' \leq l)\big) \\ T[-\infty], & \text{otherwise} \end{cases} \tag{6}$$

3.2 Validity and Skolem extraction

Skolemization (i.e., introducing Skolem functions) 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 algorithm AE-VAL for deciding validity and Skolem extraction assumes that a formula Ψ can be transformed into the form $\exists \vec{y} \,.\, \Psi(\vec{x}, \vec{y}) \equiv S(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$,

Algorithm 4: 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 $y_{j+1} = f_{j+1}(\vec{x}), \ldots, y_n = f_n(\vec{x})$.
Data: Factored formula $\pi_j(\vec{x}, y_j) = L_{\pi_j} \wedge U_{\pi_j} \wedge E_{\pi_j} \wedge N_{\pi_j}$.
Output: Local Skolem function $y_j = f_j(\vec{x})$.

```
for (i = n; i > j; i--) do
    ψ_j(x, y_j, ..., y_n) ← Substitute(ψ_j(x, y_j, ..., y_n), y_i, f_i(x));
π_j(x, y_j) ← ψ_j(x, y_j, ..., y_n);
if (E_{π_j} ≠ ∅) then return E_{π_j};
π_j(x, y_j) ← Merge(L_{π_j}, MAX, π_j(x, y_j));
π_j(x, y_j) ← Merge(U_{π_j}, MIN, π_j(x, y_j));
if (N_{π_j} = ∅) then
    if (L_{π_j} ≠ ∅ ∧ U_{π_j} ≠ ∅) then return Rewrite(L_{π_j} ∧ U_{π_j}, MID, π_j(x, y_j));
    if (L_{π_j} = ∅) then return Rewrite(U_{π_j}, LT, π_j(x, y_j));
    if (U_{π_j} = ∅) then return Rewrite(L_{π_j}, GT, π_j(x, y_j));
else return Rewrite(L_{π_j} ∧ U_{π_j} ∧ N_{π_j}, FMID, π_j(x, y_j));
```

where $S(\vec{x})$ has only existential quantifiers, and $T(\vec{x}, \vec{y})$ is quantifier-free. AE-VAL partitions the formula, and searches for a witnessing local Skolem relation of each partition. AE-VAL iteratively constructs a set of MBPs $\{T_i(\vec{x})\}$, each of which is connected with a so called local Skolem relation $\phi_i(\vec{x}, \vec{y})$, such that $\phi_i(\vec{x}, \vec{y}) \implies (T_i(\vec{x}) \iff T(\vec{x}, \vec{y}))$ (i.e., that make the corresponding projections equisatisfiable with T). While the pseudocode of AE-VAL is shown in Alg. 3, we refer the reader to [19] for more detail.

A Skolem relation $Sk_{\vec{y}}(\vec{x}, \vec{y})$ by AE-VAL maps each model of $S(\vec{x})$ to a corresponding model of $T(\vec{x}, \vec{y})$. Intuitively, ϕ_i maps each model of $S \wedge T_i$ to a model of T. Thus, in order to define the Skolem relation $Sk_{\vec{y}}(\vec{x}, \vec{y})$ it is enough to match each ϕ_i against the corresponding T_i:

$$Sk_{\vec{y}}(\vec{x}, \vec{y}) \equiv \begin{cases} \phi_1(\vec{x}, \vec{y}) & \text{if } T_1(\vec{x}) \\ \phi_2(\vec{x}, \vec{y}) & \text{else if } T_2(\vec{x}) \\ \cdots & \text{else } \cdots \\ \phi_n(\vec{x}, \vec{y}) & \text{else } T_n(\vec{x}) \end{cases} \tag{7}$$

3.3 Refining Skolem Relations into Skolem Functions

Since AE-VAL is an extension of the MBP-algorithm mentioned in Sect. 3.1, each ϕ_i (in (7)) is constructed from the substitutions made in T to produce T_i. Furthermore, each MBP in AE-VAL is constructed 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 in a Skolem relation are the enemies of program synthesis. Indeed, the final implementation should contain assignments to each existentially

quantified variable, which for the current algorithm is difficult to get. The Skolem relation provided by AE-VAL should be post-processed to get rid of inequalities. We formalize this procedure as finding a Skolem function $f_j(\vec{x})$ for each $y_j \in \vec{y}$, such that $(y_j = f_j(\vec{x})) \Longrightarrow \exists y_{j+1}, \ldots, y_n \,.\, \psi_j(\vec{x}, y_j, \ldots, y_n)$.

The key idea is presented in Alg. 4. The algorithm is applied separately for each $y_j \in \vec{y}$, starting from y_n to y_1. That is, for iteration j, the previous runs of the algorithm already delivered Skolem functions $f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$ for variables $y_{j+1}, \ldots, y_n$. Thus, the first step of the algorithm is to substitute each appearance of variables $y_{j+1}, \ldots, y_n$ in ψ_j by $f_n(\vec{x}), \ldots, f_{j+1}(\vec{x})$.

Once formula $\psi_j(\vec{x}, y_j, \ldots, y_n)$ is rewritten to form $\pi_j(\vec{x}, y_j)$ (line 3), the algorithm starts looking for a function $f_j(\vec{x})$, such that $y_j = f_j(\vec{x})$. In other words, it aims at constructing a graph of a function that is embedded in a relation.

Definition 3. *Given a variable y_j, a relation $\pi_j(\vec{x}, y_j)$, and a set $C(\pi_j)$ of linear combinations over $\vec{x}$ appeared in $\pi_j(\vec{x}, y_j)$, let us denote the following groups of conjuncts, π_j is composed from:*

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

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

In the rest of the section, we present several primitives needed to construct $y_j = f_j(\vec{x})$ out of $\pi_j(\vec{x}, y_j)$. For simplicity, we omit some straightforward details on dealing with non-strict inequalities consisting in M_{π_j} and V_{π_j} since they are similar strict inequalities consisting 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.

Lemma 1. *After all substitutions at line 3 of Alg. 4, each $\psi_j(\vec{x}, y_j, \ldots, y_n)$ is a conjunction of the form $L_{\pi_j} \wedge U_{\pi_j} \wedge M_{\pi_j} \wedge V_{\pi_j} \wedge E_{\pi_j} \wedge N_{\pi_j}$.*

Proof. Follows directly from (6). □

The procedure to extract $y_j = f_j(\vec{x})$ out of $\pi_j(\vec{x}, y_j)$ proceeds by analyzing terms in L_{π_j}, U_{π_j}, M_{π_j}, V_{π_j}, E_{π_j} and N_{π_j}. If 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 it from the following primitives.

Definition 4. *Let $l(\vec{x})$ and $u(\vec{x})$ be two linear terms, then operators MAX, MIN, MID, LT, GT are defined as follows:*

$$MAX(l,u)(\vec{x}) \triangleq ite(l(\vec{x}) < u(\vec{x}), u(\vec{x}), l(\vec{x})) \quad MIN(l,u)(\vec{x}) \triangleq ite(l(\vec{x}) < u(\vec{x}), l(\vec{x}), u(\vec{x}))$$

$$LT(u)(\vec{x}) \triangleq u(\vec{x}) - 1 \qquad GT(l)(\vec{x}) \triangleq l(\vec{x}) + 1$$

$$MID(l,u)(\vec{x}) \triangleq \frac{l(\vec{x}) + u(\vec{x})}{2}$$

Lemma 2. *If L_{π_j} consists of $n > 1$ conjuncts then it is equivalent to $y_j > MAX(l_1, MAX(l_2, \ldots MAX(l_{n-1}, l_n)))(\vec{x})$. If U_{π_j} consists of $n > 1$ conjuncts then it 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 5-6) 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 3. *If L_{π_j}, U_{π_j} consist of one conjunct each, and E_{π_j} and N_{π_j} are empty then the Skolem can be rewritten into $y_j = MID(l, u)(\vec{x})$.*

This primitive is applied (line 8) in case if 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 still N_{π_j} is empty (lines 9-10) the following primitive is applied:

Lemma 4. *If L_{π_j} consists of one conjunct and the rest of U_{π_j}, E_{π_j} and N_{π_j} are empty then the Skolem can be rewritten into $y_j = GT(l)(\vec{x})$. If U_{π_j} consists of one conjunct and the rest of L_{π_j}, E_{π_j} and N_{π_j} are empty then the Skolem can be rewritten into $y_j = LT(l)(\vec{x})$.*

Finally, the cases when N_{π_j} is not empty, should be handled separately. For this, we introduce another higher-order function *FMID* that for the given l, u and h and each $\vec{x}$ outputs either $MID(l, u)$ or $MID(l, MID(l, u))$ in case if $MID(l, u)$ is equal to h.

Lemma 5. *If each of L_{π_j}, U_{π_j} and N_{π_j} consist of one conjunct and E_{π_j} and N_{π_j} are empty then the Skolem can be rewritten into $y_j = FMID(l, u, h)(\vec{x})$, where*

$$\begin{aligned} FMID(l,u,h)(\vec{x}) = ite(&MID(l,u)(\vec{x}) = h(\vec{x}),\\ &MID(l, MID(l,u))(\vec{x}),\\ &MID(l,u)(\vec{x})) \end{aligned}$$

For bigger number of conjuncts of N_{π_j}, the Skolem gets rewritten in a similar way recursively.

Lemma 6. *If each of L_{π_j} and N_{π_j} consist of one conjunct, and E_{π_j} and U_{π_j} are empty then the Skolem can be rewritten into $y_j = FMID(l, GT(l))(\vec{x})$. If each of U_{π_j}, N_{π_j} consist of one conjunct, and the rest of M_{π_j},V_{π_j}, E_{π_j} and L_{π_j} are empty then the Skolem can be rewritten into $y_j = FMID(LT(u), u)(\vec{x})$.*

Theorem 2 (Soundness). *Iterative application of Alg. 4 to all variables $y_n, \ldots, y_1$ returns a local Skolem function to be used in* (7).

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

4 Implementation

We develop JKindSynt, our synthesis algorithm on top of JKind [21], a Java-implementation of the KIND model checker [26]. Each model is described using the Lustre Specification language, which is used as an intermediate language to formally verify contracts in the Assume-Guarantee Reasoning (AGREE) framework [12]. Internally, JKind is using two parallel engines (for *BaseCheck(n) 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 [14] which provides state of the art support for reasoning over quantifiers and incremental search.

Provided with the assumption that the check queries are satisfiable according to the results received by the SMT solver, we proceed to construct a collection of Skolem functions using the AE-VAL Skolemizer[7]. AE-VAL uses LRA as a background logic, and thus casts all numeric variables to Reals and provides the Skolem relation over Reals as well.[8] In future work, we plan to enhance AE-VAL for Linear Integer Arithmetic (LIA) to soundly support Skolem relation over Integers.

The final step of our implementation involved the creation of a specific purpose translation tool, which we currently call SMTLib2C[9]. The collection of the Skolem functions is given as an input to the compiler, which in turn uses them to generate implementations in the C language. We intend to further improve the compiler's efficiency as part of an individual future work.

5 Experimental Results

We synthesized implementations for 46 Lustre models[10] [27], including the running example from Fig. 2. The original models already contained an implementation, which provided us with a complete test benchmark suite, since we were able to compare the synthesized implementations to handwritten programs. We compared the C implementations by JKindSynt against the original models, after they had been translated to C using the LustreV6 compiler [31].

Fig. 3 shows the overhead of JKindSynt. The overhead is at expected levels for the majority of the models, with a few outstanding exceptions where it has a significant impact to the overall performance.

[7] More info about AE-VAL can be found at `http://www.inf.usi.ch/phd/fedyukovich/niagara`.

[8] To increase precision, of the realizability checks over LIA, JKind has an option to use Z3 directly.

[9] The source code is available at `https://github.com/andrewkatis/SMTLib2C`

[10] The models are part of a larger collection that can be found at https://tinyurl.com/gt4geqz

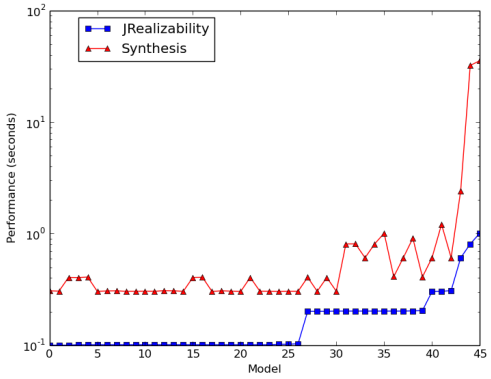


Fig. 3: Overhead of synthesis to realizability checking

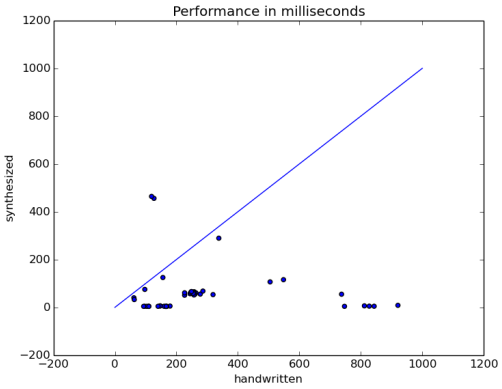


Fig. 4: Performance of synthesized and handwritten implementations

Fig. 4 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 one of the 46 models, with the x axis being the performance of the handwritten program, while the y axis reports the corresponding performance of the synthesized implementation. For the two most complex models in the benchmark suite, the synthesized implementations underperform when compared to 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 models in the experiment set, the synthesized programs perform better with a noticeable gap. We attribute these results mainly due to the simplicity of the requirements

expressed in majority of the models, as all of them were proved realizable for $k = 0$ by JKind, except for the running example, which was proved for $k = 1$.

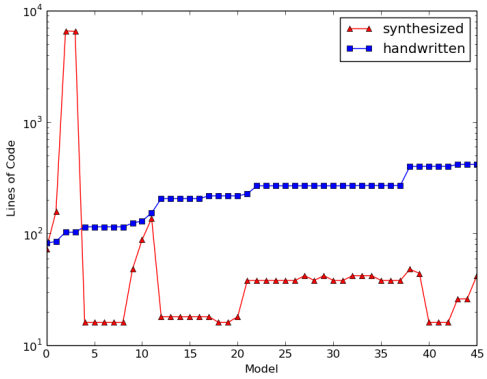


Fig. 5: Lines of code of synthesized and handwritten implementations

Fig. 5 provides another interesting, as well as important metric in our experiments, which is the lines of code in each pair of implementations. Here, we can see the direct effect of the specification complexity to the size of the Skolem functions generated by AE-VAL. Two of the three synthesized programs are also the ones that underperformed their handwritten counterparts in terms of time. Since the majority of the models contained simple requirements, the overall size of the synthesized implementation remained well below LustreV6-programs.

6 Related Work

Research in the field of program synthesis attributes its origins in the 1970s, when Zohar Manna and Richard Waldinger first introduced a synthesis procedure using theorem proving. [41]. Almost two decades later, Amir Pnueli and Roni Rosner were the first to propose a way to synthesize implementations for temporal specifications [43]. This work also involved the first formal definition of a reactive system's realizability, defined by the authors using the term implementability.

Since then, a vast variety of techniques have been developed. Efficient algorithms were proposed for subsets of propositional LTL [35,50,17,9] simple LTL formulas [6,28,49], as well as other temporal logics [3,42,29], such as SIS [1]. Component-based approaches have also been explored in [8,13].

Sumit Gulwani in 20120 published a survey on which he discribed the potential future directions of program synthesis research [25]. The approaches that have been proposed are many, and differ on many aspects, either in terms of

the specifications that are being exercised, or the reasoning behind the synthesis algorithm itself. Template-based synthesis [47] is focused on the exploration of programs that satisfy a specification that is refined after each iteration, following the basic principles of deductive synthesis. 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 [20]. 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. Recently published work on extending SMT solvers with counterexample-guided synthesis shows that they can eventually be used as an alternative to solving the problem under certain domains of arithmetic [45]. Reactive synthesis has also been explored in the context involving propositional formulas for safety specifications [5]. Finally, functional synthesis is used in applications where only a partial implementation exists, and the user needs an automated way to complete the missing parts of the program [36].

A rather important contribution in the area is the recently published work by Leonid Ryzhyk and Adam Walker [46], where they share their experience in developing and using a reactive synthesis tool for controllers in an industrial environment. While the authors emphasize that the research on program synthesis is still at a very early stage for the technique to be essential to industrial applications, they note its potential advantages in terms of improving the overall development cycle of software.

To the best of our knowledge our work is the first complete attempt on providing a synthesis algorithm for an assume-guarantee framework, using infinite theories. We take advantage of a sophisticated solver that is able to reason about the validity of the intermediate formulas that construct a k-inductive proof, as well as provide witnesses for these formulas through the use of Skolem functions. 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.

7 Future Work

The meaningful results of our work so far on the synthesis from Assume-Guarantee contracts have also provided a solid ground towards extending and improving the involved algorithms in the future. A particularly important milestone is to eventually switch to a more efficient algorithm, where we endorse the core idea of Property Directed Reachability [16,7], using efficient ways to further enhance the algorithm's performance through the use of implicit abstractions [11] to further reduce the search space of the algorithm. This will also help our original work on realizability checking, by improving the unsoundness of our unrealizable results. Another promising idea here is the use of Inductive Validity Cores (IVCs) [24], whose main purpose is to effectively pinpoint the absolutely necessary model elements in a generated proof. We can potentially use the information provided by IVCs as a preprocessing tool to reduce the size of the original specification, and

hopefully the complexity of the realizability proof. Of course, a few optimizations can be further implemented in terms of AE-VAL's specific support on proofs of realizability and finally, a very important subject is the further improvement of the compiler that we created to translate the Skolem functions into C implementations, by introducing optimizations like common subexpression elimination.

Another important goal is that of supporting additional theories, and primarily LIA, which is currently not fully supported by AE-VAL's model based projection technique. Finally, another potential optimization that could effectively reduce the algorithm's complexity is the further simplification of the transition relation that we are currently using, by reducing its complicated form through the mapping of common subexpressions on different conditional branches. This will also have a direct impact on the skolem relations retrieved by AE-VAL, reducing their individual size and improving, thus, the final implementation in terms of readability as well as its usability as an intermediate representation to the preferred target language.

8 Conclusion

Grigory: TODO

Acknowledgments

This work was funded by DARPA and AFRL under contract 4504789784 (Secure Mathematically-Assured Composition of Control Models), and by NASA under contract NNA13AA21C (Compositional Verification of Flight Critical Systems), and by NSF under grant CNS-1035715 (Assuring the safety, security, and reliability of medical device cyber physical systems).

References

1. Aziz, A., Balarin, F., Braton, R., Sangiovanni-Vincentelli, A.: Sequential Synthesis using SIS. Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'95) pp. 612–617 (1995)
2. Balabanov, V., Jiang, J.R.: Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In: CAV. LNCS, vol. 6806, pp. 149–164 (2011)
3. Beneš, N., Černá, I., Štefaňák, F.: Factorization for component-interaction automata. In: SOFSEM 2012: Theory and Practice of Computer Science, pp. 554–565. Springer (2012)
4. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: POPL. pp. 221–234. ACM (2014)
5. Bloem, R., Egly, U., Klampfl, P., Könighofer, R., Lonsing, F., Seidl, M.: Satisfiability-based methods for reactive synthesis from safety specifications. arXiv preprint arXiv:1604.06204 (2016)
6. Bohy, A., BruyÃÍre, V., Filiot, E., Jin, N., Raskin, J.F.: Acacia+, a tool for LTL Synthesis. Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12) pp. 652–657 (2012)

7. Bradley, A.: SAT-based model checking without unrolling. VMCAI (2011)
8. Chatterjee, K., Henzinger, T.A.: Assume-Guarantee Synthesis. Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07) pp. 261–275 (2007)
9. Cheng, C.H., Hamza, Y., Ruess, H.: Structural synthesis for gxw specifications. arXiv preprint arXiv:1605.01153 (2016)
10. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Parameter synthesis with IC3. In: FMCAD. pp. 165–168. IEEE (2013)
11. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Ic3 modulo theories via implicit predicate abstraction. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 46–61. Springer (2014)
12. Cofer, D.D., Gacek, A., Miller, S.P., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Goodloe, A.E., Person, S. (eds.) Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012). vol. 7226, pp. 126–140. Springer-Verlag, Berlin, Heidelberg (April 2012)
13. Damm, W., Finkbeiner, B., Rakow, A.: What you really need to know about your neighbor
14. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
15. Dutertre, B.: Solving Exists/Forall Problems With Yices. In: SMT Workshop (2015), extended abstract
16. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Formal Methods in Computer-Aided Design (FMCAD), 2011. pp. 125–134. IEEE (2011)
17. Ehlers, R.: Symbolic bounded synthesis. In: International Conference on Computer Aided Verification. pp. 365–379. Springer (2010)
18. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Ae-val: Horn clause-based skolemizer for $\forall\exists$-formulas
19. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated discovery of simulation between programs. In: Logic for Programming, Artificial Intelligence, and Reasoning. pp. 606–621. Springer (2015)
20. Flener, P., Partridge, D.: Inductive programming. Automated Software Engineering 8(2), 131–137 (2001)
21. Gacek, A.: JKind – an infinite-state model checker for safety properties in Lustre. `http://loonwerks.com/tools/jkind.html` (2016)
22. Gacek, A., Katis, A., Whalen, M.W., Backes, J., Cofer, D.: Towards realizability checking of contracts using theories. In: NASA Formal Methods, pp. 173–187. Springer (2015)
23. Gascón, A., Tiwari, A.: A Synthesized Algorithm for Interactive Consistency. In: NFM. LNCS, vol. 8430, pp. 270–284 (2014)
24. Ghassabani, E., Gacek, A., Whalen, M.W.: Efficient generation of inductive validity cores for safety properties. In: FSE2016: ACM Sigsoft International Symposium on the Foundations of Software Engineering (2016)
25. Gulwani, S.: Dimensions in program synthesis. In: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming. pp. 13–24. ACM (2010)
26. Hagen, G.: Verifying safety properties of Lustre programs: an SMT-based approach. Ph.D. thesis, University of Iowa (December 2008)
27. Hagen, G., Tinelli, C.: Scaling up the formal verification of lustre programs with smt-based techniques. In: Formal Methods in Computer-Aided Design, 2008. FMCAD '08. pp. 1–9 (Nov 2008)

28. Hagihara, S., Ueno, A., Tomita, T., Shimakawa, M., Yonezaki, N.: Simple synthesis of reactive systems with tolerance for unexpected environmental behavior. In: Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering. pp. 15–21. ACM (2016)
29. Hamza, J., Jobstmann, B., Kuncak, V.: Synthesis for Regular Specifications over Unbounded Domains. Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design pp. 101–109 (2010)
30. Heule, M., Seidl, M., Biere, A.: Efficient Extraction of Skolem Functions from QRAT Proofs. In: FMCAD. pp. 107–114. IEEE (2014)
31. Jahier, E., Raymond, P., Halbwachs, N.: The Lustre V6 Reference Manual, `http://www-verimag.imag.fr/Lustre-V6.html`
32. John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem Functions for Factored Formulas. In: FMCAD. pp. 73–80. IEEE (2015)
33. Katis, A., Gacek, A., Whalen, M.W.: Machine-checked proofs for realizability checking algorithms. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 110–123. Springer (2015)
34. Katis, A., Whalen, M.W., Gacek, A.: Towards synthesis from assume-guarantee contracts involving infinite theories: A preliminary report. arXiv preprint arXiv:1602.00148 (2016)
35. Klein, U., Pnueli, A.: Revisiting Synthesis of GR(1) Specifications. Proceedings of the 6th International Conference on Hardware and Software: Verification and Testing (HVC'10) pp. 161–181 (2010)
36. Kneuss, E., Kuncak, V., Kuraj, I., Suter, P.: On integrating deductive synthesis and verification systems. arXiv preprint arXiv:1304.5661 (2013)
37. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. In: Computer Aided Verification. vol. 8559, pp. 17–34. Springer (2014)
38. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. ACM Sigplan Notices 45(6), 316–329 (2010)
39. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. International Journal on Software Tools for Technology Transfer 15(5-6), 455–474 (2013)
40. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36(5), 450–462 (1993)
41. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Communications of the ACM 14(3), 151–165 (1971)
42. Monmege, B., Brihaye, T., Estiévenart, M., Ho, H.M., ULB, G.G., Sznajder, N.: Real-time synthesis is hard! In: Formal Modeling and Analysis of Timed Systems: 14th International Conference, FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings. vol. 9884, p. 105. Springer (2016)
43. Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module. Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'89) pp. 179–190 (1989)
44. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. pp. 179–190. ACM Press (1989)
45. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in smt
46. Ryzhyk, L., Walker, A.: Developing a practical reactive synthesis tool: Experience and lessons learned
47. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verification and program synthesis. International Journal on Software Tools for Technology Transfer 15(5-6), 497–518 (2013)

48. The Coq Development Team: The Coq Proof Assistant Reference Manual. INRIA, 8.4 edn. (2012-2014)
49. Tini, S., Maggiolo-Schettini, A.: Compositional Synthesis of Generalized Mealy Machines. Fundamenta Informaticae 60(1-4), 367–382 (2003)
50. Tomita, T., Ueno, A., Shimakawa, M., Hagihara, S., Yonezaki, N.: Safraless ltl synthesis considering maximal realizability. Acta Informatica pp. 1–38 (2016)

A Example in more detail

Here we consider our example from Fig. 2 and demonstrate one iteration of the synthesis procedure. In particular the $\forall\exists$-formula of *ExtendCheck* is as follows:

$$
\begin{aligned}
bias_0 &= ite(init, 0, ite(x_0, 1, -1) + bias_{-1}) \wedge \\
bias\_max_0 &= ite(init, false, ((bias_0 \geq 2) \vee (bias_0 \leq -2)) \vee bias\_max_{-1}) \wedge \\
prop_{10} &= ((state_0 = 0) \implies (bias_0 = 0)) \wedge \\
prop_{20} &= ite(init, true, ((state_{-1} = 0) \wedge x_0) \implies (state_0 = 2)) \wedge \\
prop_{30} &= ite(init, true, ((state_{-1} = 0) \wedge (\neg x_0)) \implies (state_0 = 1)) \wedge \\
prop_{40} &= (bias\_max_0 \implies (state_0 = 3)) \wedge \\
prop_{50} &= ((state_0 = 0) \vee (state_0 = 1) \vee (state_0 = 2) \vee (state_0 = 3)) \wedge \\
prop\_all_0 &= (prop_{10} \wedge prop_{20} \wedge prop_{30} \wedge prop_{40} \wedge prop_{50}) \wedge \\
&prop\_all_0 \wedge \\
bias_1 &= ite(false, 0, ite(x_0, 1, -1) + bias_0) \wedge \\
bias\_max_1 &= ite(false, false, ((bias_1 \geq 2) \vee (bias_1 \leq -2)) \vee bias\_max_0) \wedge \\
prop_{11} &= ((state_1 = 0) \implies (bias_1 = 0)) \wedge \\
prop_{21} &= ite(false, true, ((state_0 = 0) \wedge x_0) \implies (state_1 = 2)) \wedge \\
prop_{31} &= ite(false, true, ((state_0 = 0) \wedge (\neg x_0)) \implies (state_1 = 1)) \wedge \\
prop_{41} &= (bias\_max_1 \implies (state_1 = 3)) \wedge \\
prop_{51} &= ((state_1 = 0) \vee (state_1 = 1) \vee (state_1 = 2) \vee (state_1 = 3)) \wedge \\
prop\_all_1 &= (prop_{11} \wedge prop_{21} \wedge prop_{31} \wedge prop_{41} \wedge prop_{51}) \implies \\
\exists bias_2, bias\_max_2, &prop_{12}, state_2, prop_{22}, prop_{32}, prop_{42}, prop_{52}, prop\_all_2 \,. \\
bias_2 &= 0 \wedge \\
bias\_max_2 &= false \wedge \\
prop_{12} &= ((state_2 = 0) \implies (bias_2 = 0)) \wedge \\
prop_{22} &= true \wedge \\
prop_{32} &= true \wedge \\
prop_{42} &= (bias\_max_2 \implies (state_2 = 3)) \wedge \\
prop_{52} &= ((state_2 = 0) \vee (state_2 = 1) \vee (state_2 = 2) \vee (state_2 = 3)) \wedge \\
prop\_all_2 &= (prop_{12} \wedge prop_{22} \wedge prop_{32} \wedge prop_{42} \wedge prop_{52}) \wedge \\
&prop\_all_2
\end{aligned}
$$

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

$$
\begin{aligned}
&(x1 \wedge (-1 = bias_0)) \vee ((\neg x_1) \wedge (1 = bias_0)) \wedge \\
&\neg bias\_max_0 \wedge \\
&(\neg(state_0 = 0)) \vee (\neg x_1) \wedge (\neg(state_0 = 0)) \vee x_1
\end{aligned}
$$

and the following local Skolem function:

$$state_2 = 0 \qquad\qquad bias_2 = 0$$

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.