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. □

1 Introduction

Formal verification is a well-established area of research, with ever increasing popularity, in an attempt to provide better tools to software engineers during the design and testing phases of a project and effectively reduce its overall development cost. A particularly interesting problem in formal verification is that of program synthesis, where researchers try to construct efficient algorithms that

can automatically generate code which is guaranteed to behave correctly based on the information provided by the user through formal or informal requirements.

While the problem of synthesis has been explored in a significant amount of diverse contexts, in this paper we particularly focus on the automated generation of implementations for the leaf-level components of embedded systems, using safety properties that are expressed in the form of an Assume-Guarantee contract. In recent work [16], we introduced a novel synthesis procedure that, given a contract written in AADL, can provide an implementation that is able to react to uncontrolled inputs provided by the system's environment, while satisfying the restrictions specified in the contract assumptions and guarantees. The synthesis algorithm is an extension of our previous work on solving the problem of realizability modulo infinite theories [7], using a model checking algorithm that has been formally verified in terms of its soundness for realizable results [15]. Given the inductive proof of realizability, we take advantage of a sophisticated skolemizer for $\forall\exists$ formulas, named AE-VAL [5], that is able to provide us with witnesses of strategies that a synthesized implementation can follow at each step of execution. In the context of this paper, we have implemented the synthesis algorithm and exercised it in terms of its performance on several models. We provide an informal proof of the algorithm's correctness regarding the implementations that it produces, and discuss our experimental results.

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 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 summary of the formal background that has already been established in previous work, regarding an algorithm that is able to generate leaf-level component implementations using only the information provided by the user through requirements expressed in the form of an Assume-Guarantee contract. Our approach mainly supports the Linear Real Arithmetic (LRA) theory, and to a certain extend the theory of integers (LIA), mainly due to the limitations imposed by the underlying machinery. We begin with a brief description of an Assume-Guarantee contract, and move on to discuss the specifics of our program synthesis procedure, which depends on our earlier work towards solving the problem of realizability checking of contracts. Finally, we enrich our formal definitions with an informal proof of the algorithm's correctness in terms of the successfully synthesized implementations.

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.

Fig. 1: Example of an Assume-Guarantee contract

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 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 [7]. 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.*

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

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 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 [15].

2.4 Program Synthesis from Proofs of Realizability

The most important outcome of the 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 BaseCheckEngine.SmtAdd(¬BaseCheck(i));
  ExtendCheckEngine.SmtAdd(¬ExtendCheck(i));
3 forever do
4 |   i++;
5 |   if (BaseCheckEngine.isSat(SmtSolve())) then return UNREALIZABLE;
6 |   SkolemsAdd(AE-VAL(BaseCheck(i));
7 |   if (ExtendCheckEngine.isUnsat(SmtSolve())) then
  |   SkolemsAdd(AE-VAL(ExtendCheck(i)));
8 |   return Skolems, REALIZABLE;
```

Algorithm 1 provides a summary of the synthesis algorithm, showing how it extends our previous work on realizability checking. During the k-induction, and for every step for which the negation of *BaseCheck(n)* is not satisfiable, we feed the corresponding $\forall\exists$-formula to AE-VAL. A formula that is not satisfiable means that its negation is valid, and as such we expect AE-VAL to strengthen *BaseCheck(n)*'s proof of validity. In addition to this proof, AE-VAL also returns a Skolem function that essentially assigns specific values to the existentially quantified variables, i.e. the output variables, in order to traverse into a new state, without violating the contract. We keep repeating this process, accumulating Skolem functions as long as the corresponding *BaseCheck(n)* is true. 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.

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[1]();
4  ...
5  READ_INPUTS();
6  SKOLEMS[K]();
7  forever do
8  |  READ_INPUTS();                       ▷ Transition using ExtendCheck witness
9  |  SKOLEMS[K]();
10 |  UPDATE_ARRAY_HISTORY();
```

With a collection of Skolems at hand, we are able to construct the skeleton of a candidate implementation as shown in Alg. 2. 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.

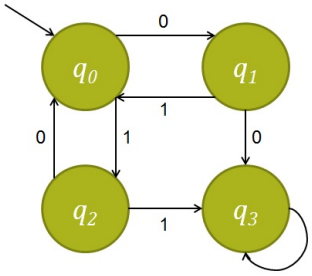
For the purposes of this paper, we developed[5] a translation tool to reconstruct the collections of Skolem functions returned by our synthesis algorithm, to C implementations, following the steps presented by Alg. 2. The performance of the resulting implementations are being presented in Section 5.

[5] You can find the translation tool at https://github.com/andrewkatis/SMTLib2C

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 [6]. 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 [21].

2.5 Running Example

To further understand the concepts described in this paper, we provide a simple running example that is representative of what happens internally during a single run of the synthesis algorithm. Figure 2 is showing a simple automaton that we will be using as the example. To the right, we can see the requirements that were written for the automaton in the Lustre language. The specification mentions one input, namely x. While the *state* variable would otherwise be considered as the output in the resulting implementation, we use a simple hack that prevents us from explicitly defining how it should be assigned, by adding it as an actual input to the specification. In addition to the previous, can see that there are 4 properties to be checked, along with a JKind-specific query on the specification's realizability. Properties *prop2* and *prop3* are used to indirectly summarize parts of the possible transitions in the automaton. Properties *prop1* and *prop4* are 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.




```
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;
  prop_all = prop1 and prop2 and prop3 and prop4;
  --%PROPERTY prop_all;
  --%REALIZABLE x;
tel;
```

Fig. 2: Automaton and Requirements for running example

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

Considering these requirements, we want an answer to whether this model is realizable. We provide the model as an input to JKind, and observe the results. The model is sufficiently simple to be declared as realizable by the tool, and thus can be further examined in terms of synthesizing a representative implementation. Using the k-inductive proof of realizability, which happens to be of length $k = 1$, we forward the two $\forall\exists$-formulas ($k = 0$ for the base check, and $k = 1$ for the inductive check) to be skolemized by AE-VAL. From this process, we receive two Skolem functions, that effectively describe possible assignments to the local variables of the specification, as well as the output *state* (we can effectively decide what is an output by excluding it from the list of arguments in the $-\%REALIZABLE$ query in Fig 2). We can then use these two Skolem functions to construct the final implementation following the outline provided in Alg. 2. Due to the fact that the implementation is too long to be included here (135 lines of code), we do not present its actual form[7]. The main idea involves the redefinition of each variable in the model as an array of size equal to the proof's length k, and, using 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. One of the interesting features that come from using Lustre in conjunction with a k-inductive proof is that we can possibly have properties in the model that refer to up to $k - 1$ values of each variable in the past, and as such an update process occurs in between each iteration of the infinite loop, to ensure that we only keep the k latest values of each array.

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., [1,18,12,14]). Various tasks in verification and synthesis [3,2,8] 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* [20]), 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.

[7] The implementation for the example is available at

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 [17,4].

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

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

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

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 [17]. It is based on Loos-Weispfenning (LW) quantifier-elimination method [19] 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:

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

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):

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

Algorithm 3: AE-VAL$\Big(S(\vec{x}), \exists \vec{y}\,.\,T(\vec{x},\vec{y})\Big)$, cf. [5]

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);
```

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 $\exists \vec{y}\,.\,\Psi(\vec{x},\vec{y}) \equiv S(\vec{x}) \implies \exists \vec{y}\,.\,T(\vec{x},\vec{y})$, 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 [5] 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:

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 $y_j = f_j(\vec{x})$.

1 **for** $(i = n; i > j; i{-}{-})$ **do**
2 $\quad \psi_j(\vec{x}, y_j, \ldots, y_n) \leftarrow$ Substitute$(\psi_j(\vec{x}, y_j, \ldots, y_n), y_i, f_i(\vec{x}))$;
3 $\pi_j(\vec{x}, y_j) \leftarrow \psi_j(\vec{x}, y_j, \ldots, y_n)$;
4 **if** $(E_{\pi_j} \neq \varnothing)$ **then return** E_{π_j};
5 $\pi_j(\vec{x}, y_j) \leftarrow$ Merge$(L_{\pi_j}, MAX, \pi_j(\vec{x}, y_j))$;
6 $\pi_j(\vec{x}, y_j) \leftarrow$ Merge$(U_{\pi_j}, MIN, \pi_j(\vec{x}, y_j))$;
7 **if** $(N_{\pi_j} = \varnothing)$ **then**
8 $\quad$ **if** $(L_{\pi_j} \neq \varnothing \wedge U_{\pi_j} \neq \varnothing)$ **then return** Rewrite$(L_{\pi_j} \wedge U_{\pi_j}, MID, \pi_j(\vec{x}, y_j))$;
9 $\quad$ **if** $(L_{\pi_j} = \varnothing)$ **then return** Rewrite$(U_{\pi_j}, LT, \pi_j(\vec{x}, y_j))$;
10 $\quad$ **if** $(U_{\pi_j} = \varnothing)$ **then return** Rewrite$(L_{\pi_j}, GT, \pi_j(\vec{x}, y_j))$;
11 **else return** Rewrite$(L_{\pi_j} \wedge U_{\pi_j} \wedge N_{\pi_j}, FMID, \pi_j(\vec{x}, y_j))$;

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

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 $(y_j = f_j(\vec{x})) \implies \exists y_{j+1}, \ldots, y_n \,.\, \psi_j(\vec{x}, y_j, \ldots, y_n)$.

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 1 (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

The synthesis algorithm presented in this work is currently a feature provided by JKind [6], a re-implementation of the KIND model checker [10] in Java. 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 [**?**]. Internally, JKind is using two parallel engines in order to construct a k-inductive proof for the property of interest. The first order formulas that are being constructed are then feeded to the Z3 SMT solver [**?**] 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. Andreas: Add AE-VAL related implementation details here.

The final step of our implementation involved the creation of a specific purpose translation tool, which is currently called SMTLib2C [8]. The translation tool is still at a very primitive state in terms of optimizations, a goal that we eventually want to address as part of our future work.

5 Experimental Results

To evaluate our work for this paper, we synthesized implementations for 46 Lustre models [9] [11], including the running example. 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.

To effectively compare the synthesized programs, we developed a compiler from primitive scratch files that contain the collection of Skolem functions describing the implementation, to C programs. We then compared these C implementations against the original models, after they had been translated to C using the LustreV6 compiler [13].

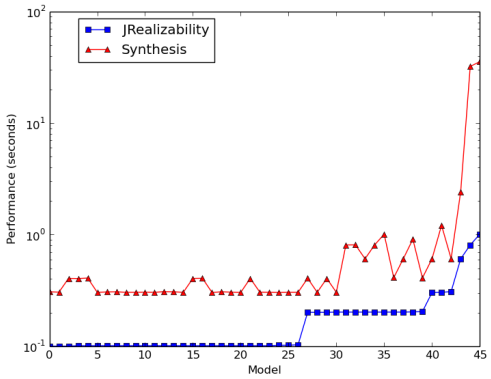


Fig. 3: Overhead of synthesis to realizability checking

Figure 3 shows the overhead of our extension to JKind's realizability checking algorithm to support synthesis. The overhead is at expected levels for the majority of the models, with a few outstanding exceptions where it has a significant

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

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

impact to the overall performance. A particularly interesting way to improve upon this is by switching to a more sophisticated algorithm, where we endorse the core idea of Property Directed Reachability in terms of finding a proof of realizability, in conjunction with AE-VAL's skolemization procedure.

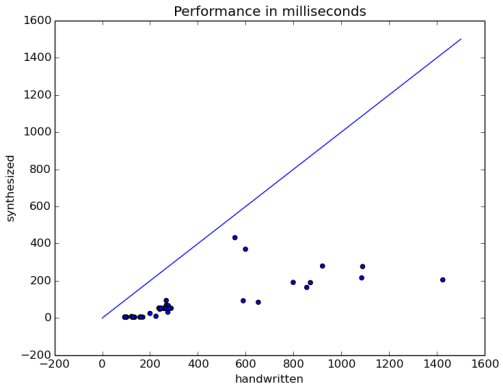


Fig. 4: Performance of synthesized and handwritten implementations

Figure 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. In every case of this benchamrk suite, the synthesized implementations outperform the programs generated by LustreV6. We attribute this fact mainly due to the simplicity of the requirements expressed in the files, as all of them were proved realizable for $k = 0$ by JKind, except for the running example , which was proved for $k = 1$. In addition to the above, we have to take into consideration that LustreV6's C code generation feature is still documented as work in progress, and thus might lack optmizations that could possibly help reduce the performance gap.

Figure 5 provides another interesting, as well as important metric in our experiments, which is the lines of code in each pair of implementations. In the majority of the models that we used, the overall size of the synthesized implementation remained well below LustreV6's programs. Despite this, a few notable outliers still exist, where the actual size of the synthesized implementation is bigger, with two models exceeding an order of magnitude when compared to their handwritten counterparts. These two particular models were also the most complex ones in terms of the specification in our test suite, and as a result the corresponding Skolem functions are also very big in terms of size.

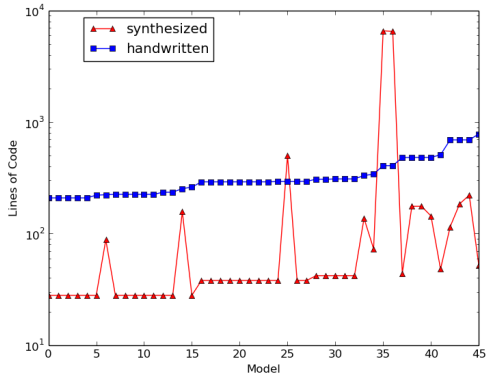


Fig. 5: Lines of code of synthesized and handwritten implementations

For future work, we hope to tackle such cases on three different frontiers. The first is again the use of a better algorithm that can effectively reduce the size of the transition relation used during the realizability checking algorithm. Another interesting idea here is the use of Inductive Validity Cores (IVCs) [9], 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.

6 Related Work

7 Future Work

While our current approach to program synthesis has been shown to be effective in the contracts that we have exercised, there are yet a lot of interesting ways to extend and optimize the underlying algorithm, to yield better results in the future. An important extension is that of supporting additional theories such as integers, which is currently not supported by AE-VAL's model based projection technique. To combat the lack of soundness on unrealizable results, and thus missing potential synthesized implementations, we will be developing a new algorithm that is mainly based on the idea of generating inductive invariants, much like the way that is presented in Property Directed Reachability algorithms. 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.

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. Balabanov, V., Jiang, J.R.: Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In: CAV. LNCS, vol. 6806, pp. 149–164 (2011)
2. 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)
3. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Parameter synthesis with IC3. In: FMCAD. pp. 165–168. IEEE (2013)
4. Dutertre, B.: Solving Exists/Forall Problems With Yices. In: SMT Workshop (2015), extended abstract
5. 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)
6. Gacek, A.: JKind – an infinite-state model checker for safety properties in Lustre. `http://loonwerks.com/tools/jkind.html` (2016)
7. 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)
8. Gascón, A., Tiwari, A.: A Synthesized Algorithm for Interactive Consistency. In: NFM. LNCS, vol. 8430, pp. 270–284 (2014)
9. 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)
10. Hagen, G.: Verifying safety properties of Lustre programs: an SMT-based approach. Ph.D. thesis, University of Iowa (December 2008)
11. 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)
12. Heule, M., Seidl, M., Biere, A.: Efficient Extraction of Skolem Functions from QRAT Proofs. In: FMCAD. pp. 107–114. IEEE (2014)
13. Jahier, E., Raymond, P., Halbwachs, N.: The Lustre V6 Reference Manual, `http://www-verimag.imag.fr/Lustre-V6.html`
14. John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem Functions for Factored Formulas. In: FMCAD. pp. 73–80. IEEE (2015)

15. 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)
16. 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)
17. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. In: Computer Aided Verification. vol. 8559, pp. 17–34. Springer (2014)
18. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT 15(5-6), 455–474 (2013)
19. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36(5), 450–462 (1993)
20. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. pp. 179–190. ACM Press (1989)
21. The Coq Development Team: The Coq Proof Assistant Reference Manual. INRIA, 8.4 edn. (2012-2014)