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. Grigory: There is a need to the final sentence.. something like "our C-implementations are better than the ones by 'LustreC.' □

Program synthesis is a particularly interesting area of research in artificial intelligence, and more recently in formal verification. The main idea is generate efficient implementations for systems, using the system-specific requirements as the only source of information. This is especially important for the case of embedded systems that are meant to be used as leaf-level, independent components of a bigger, more complex hierarchical architecture.

One of the very challenging problems in requirements engineering is for one to decide whether the documented requirements are good enough

to pivot the development of an implementation that is guaranteed to meet them, given any circumstance. This is mostly known as the implementability, or realizability problem. Misconceptions and conflicts between requirements may occur during this process, and might not be easy to detect without the use of sophisticated tools, meaning that the engineer can end up with specification for which an implementation does not even exist. This fact alone can impose a big overhead in the system's development cycle, both in time and in cost of development. Having a proof of realizability for our requirements though, among other interesting requirement characteristics, directly implies that we can construct an implementation for them. Furthermore, if used properly, the same proof can be actually used to construct such an implementation automatically, and thus solve the problem of program synthesis.
In the context of this paper, we propose a program synthesis algorithm for requirements written in the form of an Assume-Guarantee contract, using the Lustre specification language. The algorithm relies heavily on the proof of the contract's realizability, using an approach that is very similar to k-induction model checking, with the additional use of quantifiers. With the k-inductive proof as a guide, and a sophisticated tool for extracting Skolem functions from ∀∃-formulas, we can effectively synthesize implementations that, by definition, are guaranteed to comply with the contract. We have incorporated the main synthesis algorithm as an extension to the realizability check provided in the JKind model checker, and furthermore developed a compiler to translate these primitive implementations to the C language.

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 [13], 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 [6], using a model checking algorithm that has been formally verified in terms of its soundness for realizable results [12]. Given the inductive proof of realizability, we take advantage of a sophisticated skolemizer for ∀∃ 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 two separate case studies. 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 **??** 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.

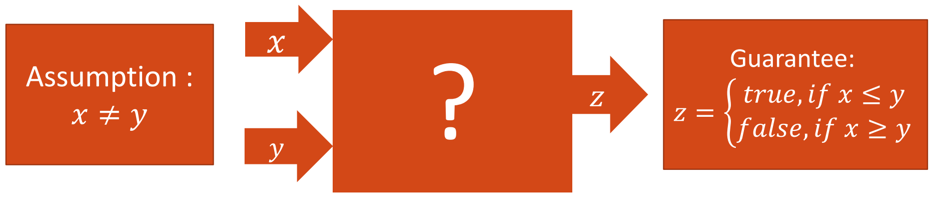
Assumption :
$x \neq y$
x
y
?
z
Guarantee:
$z = \begin{cases} true, if\ x \leq y \\ false, if\ x \geq y \end{cases}$

Fig. 1: Example of an Assume-Guarantee contract

As an illustrative example, consider the contract specified in Figure 1. 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 1 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 [6]. 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 $\mathit{Extend}_n(s)$, if any valid path of length $n-1$ starting from s can be extended in response to any input.*

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

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:

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

$$\mathit{ExtendCheck}(n) \triangleq \forall s.\mathit{Extend}_n(s) \tag{2}$$

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.

2.4 Program Synthesis from Proofs of Realizability

The most important outcome of the work on realizability is that it could be further used for solving a more complex problem of *program synthesis*, i.e., to automatically derive implementations, given the same set of requirements as for the realizability checking. Whenever a proof that a desired implementation for the given contract exists, we can use this proof in order to construct a witness implementation that satisfies the contract.

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 $\mathit{Extend}_n(s)$ 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*.

Algorithm 1: Synthesized implementation.

```
Input: Grigory: tbd
1  ASSIGN_GI_WITNESS_TO_S();      ▷ Initialize state values in an array of size $k$.
2  UPDATE_ARRAY_HISTORY();
3  READ_INPUTS();                  ▷ Perform bounded "base check" synthesis
4  BASE_CHECK_1_SOLUTION();
5  UPDATE_ARRAY_HISTORY();
6  ...
7  READ_INPUTS();
8  BASE_CHECK_K_SOLUTION();
9  UPDATE_ARRAY_HISTORY();
10 forever do
11     READ_INPUTS();              ▷ Perform recurrence from "extends" check
12     EXTEND_CHECK_K_SOLUTION();
13     UPDATE_ARRAY_HISTORY();
```

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.

Grigory: The algorithm description should be improved. Currently it mixes both, the synthesized implementation, and the synthesis procedure itself. There is an infinite loop that clearly belongs to the implementation, but the reviewers may misunderstand it. Of course, we should explicitly point that the synthesis procedure always terminates since there is a need to solve and skolemize finite number of formulas.

Thus, we can construct the skeleton of an algorithm as shown in Alg. 1. The algorithm stars (method ASSIGN_GI_WITNESS_TO_S()) with creating an array for each input and history variable up to depth k, where k is the depth at which we found a solution to our realizability algorithm. In each array, the zeroth element is the "current" value of the variable, the first element is the previous value, and the $(k-1)$'th value is the $(k-1)$-step previous value.

The algorithm then generates witnesses for each of the *BaseCheck* instances of successive depth 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 $(k-1)$'th value is simply forgotten), and READ_INPUTS() reads the current values of inputs into the zeroth element of the input variable arrays. Once the history is entirely initialized using the *BaseCheck* witness values, the algorithm generates a 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.

Andreas: Add proof of correctness here

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., [1,15,9,11]). Various tasks in verification and synthesis [3,2,7] 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* [17]), 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.

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 [14,4].

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 [14]. It is based on Loos-Weispfenning (LW) quantifier-elimination method [16] 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 2: AE-VAL$\Big(S(\vec{x}), \exists \vec{y} \,.\, T(\vec{x}, \vec{y})\Big)$

```
Input: $S(\vec{x}), \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$
Output: return value $\in \{$VALID, 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($S(\vec{x})$);
2  $i \leftarrow 0$;
3  forever do
4      $i$++;
5      if (isUNSAT(SmtSolve())) then return VALID;
6      SmtPush();
7      SmtAdd($T(\vec{x}, \vec{y})$);
8      if (isUNSAT(SmtSolve())) then return INVALID;
9      $m_i \leftarrow$ SmtGetModel();
10     $(T_i, \phi_i(\vec{x}, \vec{y})) \leftarrow$ GetMBP($\vec{y}, m_i, T(\vec{x}, \vec{y})$));
11     SmtPop();
12     SmtAdd($\neg T_i$);
```

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

In (5), $\mathit{lits}(T)$ 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 $\mathit{LRAProj}_T$ is an implementation of the *MBP* function for (5):

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

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 3: LocalFactor($y_j, \phi(\vec{x}, \vec{y})$)
  Input: $y_j \in \vec{y}$, local Skolem relation $\phi(\vec{x}, \vec{y}) = \bigwedge_{y_j \in \vec{y}} (\psi_{y_j}(\vec{x}, y_j, \ldots, y_n))$
  Output: factor of the local Skolem refinement $y_j = f_j(\vec{x})$
  Data: known functions $f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$
1 for $(i = n; i > j; i++)$ do
2   | $\psi_{y_j}(\vec{x}, y_j, \ldots, y_n) \leftarrow$ Substitute($\psi_{y_j}(\vec{x}, y_j, \ldots, y_n), y_i, f_i(\vec{x})$);
3 return SkolemFunction$\Big(\psi_{y_j}(\vec{x}, y_j, \ldots, y_n)\Big)$;
```

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. 2, 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:

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

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 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.

For each $y_j \in \vec{y}$, our goal is to find a Skolem function $f_{y_j}(\vec{x})$, such that $(y_j = f_{y_j}(\vec{x})) \implies \exists y_{j+1}, \ldots, y_n \,.\, \psi_{y_j}(\vec{x}, y_j, \ldots, y_n)$. The idea is presented in Alg. 3. The algorithm is applied separately for each $y_j \in \vec{y}$, starting from y_n till y_1. For each y_j, assume, we already established Skolem functions $f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$ for variables $y_{j+1}, \ldots, y_n$ in the previous runs of the algorithm. First, the algorithm substitutes each appearance of variables $y_{j+1}, \ldots, y_n$ in ψ_{y_j} by $f_n(\vec{x}), \ldots, f_{j+1}(\vec{x})$. Second, the algorithm gets rid of inequalities by transforming them into equalities, thus producing a Skolem function. In the rest of the section we show several fundamental rules and properties behind this operation.

Lemma 1. *After all substitutions in line 2 of Alg. 3, each $\psi_{y_j}(\vec{x}, y_j, \ldots, y_n)$ is a conjunction of the form $L_{y_j} \wedge U_{y_j} \wedge M_{y_j} \wedge V_{y_j} \wedge E_{y_j} \wedge N_{y_j}$ where $L_{y_j} = \bigwedge_l (y_j >$*

$l(\vec{x}))$, $U_{y_j} = \bigwedge_u (y_j < u(\vec{x}))$, $M_{y_j} = \bigwedge_l (y_j \geq l(\vec{x}))$, $V_{y_j} = \bigwedge_u (y_j \leq u(\vec{x}))$, $E_{y_j} = \bigwedge_e (y_j = e(\vec{x}))$, $N_{y_j} = \bigwedge_n (y_j \neq n(\vec{x}))$.

Proof. Follows directly from (6). □

The procedure to extract a Skolem function out of a Skolem relation proceeds by analyzing terms in L_{y_j}, U_{y_j}, M_{y_j}, V_{y_j},E_{y_j} and N_{y_j}. If there is at least one conjunct $(y_j = e(\vec{x})) \in E_{y_j}$ then $(y_j = e(\vec{x}))$ itself is a Skolem function. Otherwise, the algorithm creates it from the following primitives.

Definition 3. *Let $l(\vec{x})$ and $u(\vec{x})$ be two linear functions, the higher order functions MAX, MIN, MID, LT, GT are defined as follows:*

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

Lemma 2. *If L_{y_j} consists of $n > 1$ conjuncts then it is equivalent to $y_j > MAX(l_1, MAX(l_2, \dots MAX(l_{n-1}, l_n)))(\vec{x})$. If U_{y_j} consists of $n > 1$ conjuncts then it is equivalent to $y_j < MIN(u_1, MIN(u_2, \dots MIN(u_{n-1}, u_n)))(\vec{x})$.*

Similar for M_{y_j} and for V_{y_j}.

Lemma 3. *If L_{y_j} consists of 1 conjunct and the rest of U_{y_j}, M_{y_j},V_{y_j},E_{y_j} and N_{y_j} are empty then the Skolem can be rewritten into $y_j = GT(l)(\vec{x})$.*

Similar for U_{y_j} (Skolem rewritten into $y_j = MID(u)(\vec{x})$).

Lemma 4. *If L_{y_j}, U_{y_j} consist of 1 conjunct each, and the rest of M_{y_j},V_{y_j}, E_{y_j} and N_{y_j} are empty then the Skolem can be rewritten into $y_j = MID(l,u)(\vec{x})$.*

Similar for M_{y_j} and V_{y_j}, and for combinations with L_{y_j} and U_{y_j}.

Lemma 5. *If L_{y_j}, U_{y_j} and N_{y_j} consist of 1 conjunct each and the rest of M_{y_j},V_{y_j},E_{y_j} and N_{y_j} are empty then the Skolem can be rewritten into $y_j = FMID(l,u,h)(\vec{x})$ where*

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

Similar for M_{y_j} and V_{y_j}. For bigger number of conjuncts of N_{y_j}, the Skolem gets rewritten in a similar way cascadically.

Lemma 6. *If L_{y_j}, N_{y_j} consist of 1 conjunct each, and the rest of M_{y_j},V_{y_j}, E_{y_j} and U_{y_j} are empty then the Skolem can be rewritten into $y_j = FMID(l, GT(l))(\vec{x})$.*

Similar for M_{y_j}.

Lemma 7. *If U_{y_j}, N_{y_j} consist of 1 conjunct each, and the rest of M_{y_j}, V_{y_j}, E_{y_j} and L_{y_j} are empty then the Skolem can be rewritten into $y_j = FMID(LT(u), u)(\vec{x})$.*

Similar for V_{y_j}.

4 Experimental Results

To evaluate our work for this paper, we synthesized implementations for 46 Lustre models [5] [8], 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 [10].

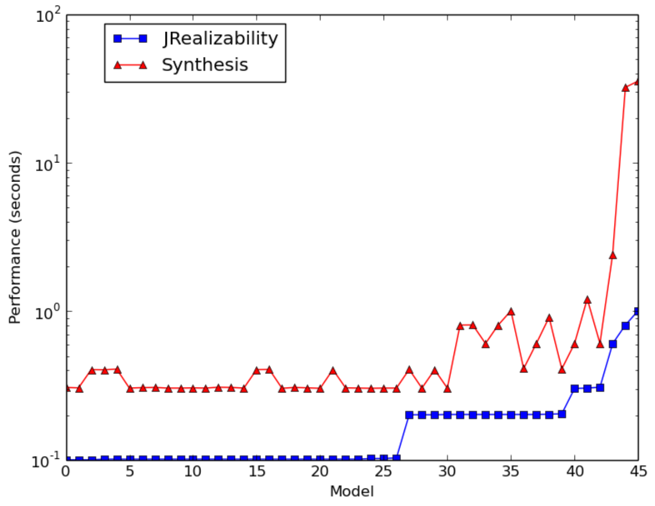

Fig. 2: Overhead of synthesis to realizability checking

Figure 2 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

[5] 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.

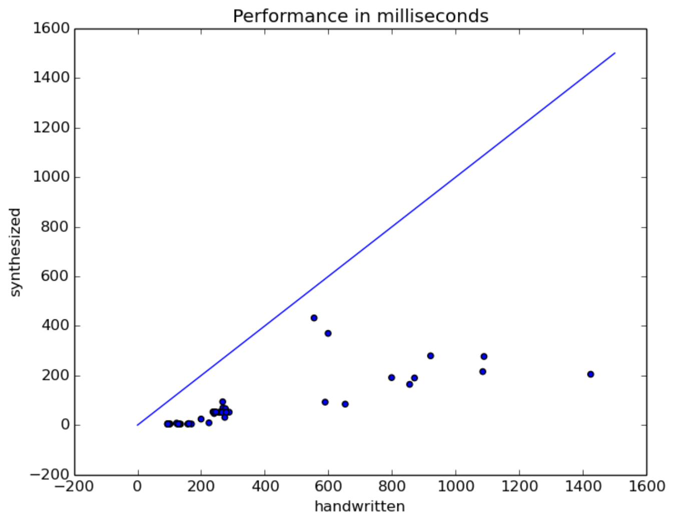

Fig. 3: Performance of synthesized and handwritten implementations

Figure 3 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.

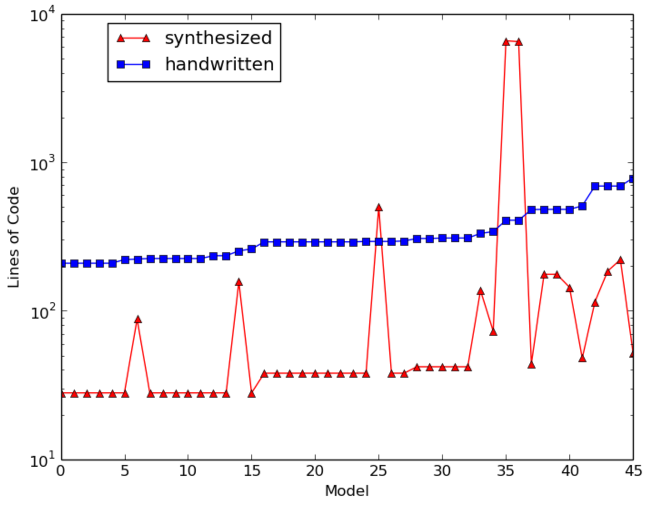

Fig. 4: Lines of code of synthesized and handwritten implementations

Figure 4 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.

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) [], 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.

5 Implementation

Grigory: moved the paragraph from section 2 here. More info about the implementation would be desirable. Some details about smtlib2 -> C conversion, maybe

The structure of the Skolem relation is simple enough to translate into a program in a mainstream language. We need implementations that are able to keep track of the current state variables, the current inputs, as well as some history about the variable values in previous states. This can easily be handled, for example, in C with the use of arrays to keep record of each variable's k last values, and the use of functions that update each variable's corresponding array to reflect the changes following a new step using the transition relation.

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., 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)
7. Gascón, A., Tiwari, A.: A Synthesized Algorithm for Interactive Consistency. In: NFM. LNCS, vol. 8430, pp. 270–284 (2014)
8. 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)
9. Heule, M., Seidl, M., Biere, A.: Efficient Extraction of Skolem Functions from QRAT Proofs. In: FMCAD. pp. 107–114. IEEE (2014)
10. Jahier, E., Raymond, P., Halbwachs, N.: The Lustre V6 Reference Manual, `http://www-verimag.imag.fr/Lustre-V6.html`
11. John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem Functions for Factored Formulas. In: FMCAD. pp. 73–80. IEEE (2015)
12. Katis, A., Gacek, A., Whalen, M.W.: Machine-checked proofs for realizability checking algorithms (2015), submitted `http://arxiv.org/abs/1502.01292`
13. 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)
14. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. In: Computer Aided Verification. vol. 8559, pp. 17–34. Springer (2014)
15. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT 15(5-6), 455–474 (2013)
16. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36(5), 450–462 (1993)
17. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. pp. 179–190. ACM Press (1989)