Synthesis from Assume-Guarantee Contracts using Skolemized Proofs of Realizability

Andreas Katis[1], Grigory Fedyukovich[2], Andrew Gacek[3], John Backes[3], 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, Washington, 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

Abstract. Grigory: I think, in the abstract, we need to be more enthusiastic about the contributions. Currently, it reads like there is a little of novelty: we extend the prior work, the approach is similar to something else, the evaluation is simple and so on. I believe, the abstract should sell the technique, and be precise about the positive evaluation points. Also, I would avoid messages like "In recent work" and "our realizability checking algorithm" since the reviewers most likely are unaware of them. Instead, I would describe a bit what is "k- inductive proof" and possibly mention that this proof is not necessarily to be given by your algorithm. This way, the synthesis algorithm will look more generic, and the paper's contributions will be stronger. Finally, the abstract should not contain the summary of the paper (i.e., the last two sentences).

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. The resulting implementations in C are further being tested and compared against the corresponding implementations provided by LustreC, a compiler from Lustre to C implementations.

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 [12], 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 [11]. Given the inductive proof of realizability, we take advantage of a sophisticated skolemizer for ∀∃ formulas, named AE-VAL [6], 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 4.1 presents our results on using the algorithm to automatically generate leaf-level component implementations for different case studies. Finally, in Section 5 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 6.

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. 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 what the system's behavior should be, using properties that precisely describe the kinds of valid outputs that it may return to its environment.

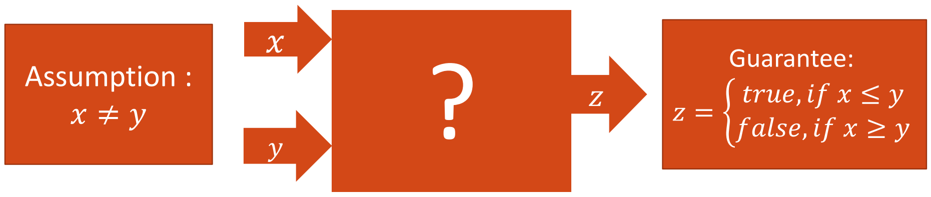
Assumption :
$x \neq y$
x
y
?
z
Guarantee:
$z = \begin{cases} true, if\ x \leq y \\ false, if\ x \geq y \end{cases}$

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 component's output is a 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 questionmark symbol. The questionmark is simply expressing the fact that during the early stages of software development, the implementation is absent or exists only partially. This is particularly important with respect to the problem of *realizability*, where we try to answer whether there exists an implementation that will satisfy the specific contract, under all circumstances. It is obvious that this is also particularly important at the harder problem of *program synthesis*, where the goal is to construct a witness of the contract's proof of realizability. In Figure 1, one can easily answer that the contract is *realizable*, and therefore a synthesis procedure should be able to provide us with an implementation. On the other hand, if we omit the contract's assumption, we can safely say that the contract is *unrealizable* as no implementation will be 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 $state \rightarrow bool$, in addition to a transition relation $T(s, i, s')$ that implements the contract and has the type $state \rightarrow inputs \rightarrow state \rightarrow bool$.

An Assume-Guarantee contract can formally defined by two sets, a set of *assumptions* and a set *guarantees*. The *assumptions* A impose constraints over the inputs, while the *guarantees* G used for the corresponding constraints over the system's outputs 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 of this paper is essentially an extension on our previous work on the realizabiility problem. Given the formal foundations above, we expressed the problem of realizability using the notion of a state being *extendable*:

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

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

The algorithm for realizability is using Definition 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 $k <= n$, while the inductive step of *ExtendCheck* tries to prove that all valid states are extendable. Therefore, we try to find the smallest n, for which the two following checks hold:

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

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

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.

2.4 Program Synthesis from the proof of Realizability

While the implemented algorithm on realizability provided us with meaningful results during the verification of several contracts, the most apparent and important outcome of this work was the fact that it could be effectively used as the basis towards solving a more complex problem, which is that of *program synthesis*. Synthesis is defined as the process of automatically deriving implementations, given a set of requirements specified by the user. Since we are able to derive a proof regarding a contract's realizability, i.e. a proof that an implementation exists for the specified contract, we can use this proof in order to construct a witness implementation that satisfies it. The limited power of SMT solvers in terms of solving formulas containing nested quantifiers immediately ruled out the prospect of using one as our primary synthesis tool. Fortunately, a work from Fedyukovich et al. [5,6] on a skolemizer for $\forall\exists$-formulas modulo theories aleviates this problem.

The skolemizer, named AE-VAL, is using the Model-Based Projection technique in [13] to check the validity for formulas of the form $\forall \vec{x}.S(\vec{x}) \Rightarrow \exists \vec{y}.T(\vec{x}, \vec{y})$, where $S(\vec{x})$ and $T(\vec{x}, \vec{y})$ are quantifier-free. If the formula is valid, a Skolem relation between the universally and existentially quantified variables is returned using a modified version of the Loos-Weispfenning quantifier elimination procedure [15], that underapproximates the existential part of the formula. The algorithm initially distributes the models of the original formula into disjoint uninterpreted partitions, with a local Skolem relation being computed for each partition in the process. From there, the use of a Horn-solver provides an interpretation for each partition, and a final global Skolem relation is produced.

The idea behind our approach to solving the synthesis problem is straightforward. Consider the checks 1 and 2 that are used in the realizability checking algorithm. Both checks require that the reachable states explored are extendable using Definition 1. The idea then is to use the definition of $Extend_n(s)$ as input to AE-VAL, which will effectively give us a witness for each of the n times that we run *BaseCheck* and a final witness for the inductive case in *ExtendCheck*. Of course, $Extend_n(s)$ as defined in 1 cannot be directly used for this purpose due to its form. This is not really an obstacle though, as we can rewrite the definition:

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

into an equivalent formula of the form $\forall \vec{x}.S(\vec{x}) \Rightarrow \exists \vec{y}.T(\vec{x}, \vec{y})$:

$$\begin{aligned}
&\forall i_1, s_1, \ldots, i_n, s_n, i. \\
&\quad A(s, i_1) \wedge G_T(s, i_1, s_1) \wedge \ldots \wedge A(s_{n-1}, i_n) \wedge G_T(s_{n-1}, i_n, s_n) \wedge A(s_n, i) \Rightarrow \\
&\qquad\qquad\qquad \exists s'.G_T(s_n, i, s')
\end{aligned} \tag{3}$$

Thus, we can construct the skeleton of an algorithm as shown in Figure 2. We begin by 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. We then generate witnesses for each of the *BaseCheck* instances of successive depth using the AE-VAL skolemizer 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 array's elements 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, we enter a recurrence loop where we use the solution of the *ExtendCheck* to describe the next value of outputs.

Andreas: Add proof of correctness here

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

```
// for each variable in I or S,
//   create an array of size k.
//   then initialize initial state values
assign_GI_witness_to_S;
update_array_history;

// Perform bounded 'base check' synthesis
read_inputs;
base_check'_1_solution;
update_array_history;
...
read_inputs;
base_check'_k_solution;
update_array_history;

// Perform recurrence from 'extends' check
while(1) {
 read_inputs;
 extend_check_k_solution;
 update_array_history;
}
```

Fig. 2. Algorithm skeleton for synthesis

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.

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., [1,14,9,10]). Various tasks in verification and synthesis [17,3,2,8] 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* [16]), 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 [13,4].

In the following, we use vector notation to denote sets of variables (and set-theoretic operators of *subset* $\vec{u} \subseteq \vec{x}$, *complement* $\vec{x}_{\vec{u}} = \vec{x} \setminus \vec{u}$, *union* $\vec{x} = \vec{u} \cup \vec{x}_{\vec{u}}$).

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

$$\textit{if } m \models T(\vec{x}, \vec{y}) \textit{ then } m \models MBP_{\vec{y}}(m, T) \tag{4}$$

$$MBP_{\vec{y}}(m, T) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y}) \tag{5}$$

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 [13]. It is based on Loos-Weispfenning (LW) quantifier-elimination method [15] 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 (6) holds:

$$\exists y \,.\, T(\vec{x}) \equiv \Big(\bigvee_{(y=e) \in lits(T)} T[e] \vee \bigvee_{(l<y) \in lits(T)} T[l + \epsilon] \vee T[-\infty] \Big) \tag{6}$$

In (6), *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 (6):

$$LRAProj_T(m) = \begin{cases} T[e], & \text{if } (y = e) \in lits(T) \wedge m \models (y = e) \\ T[l + \epsilon], & \text{else if } (l < y) \in lits(T) \wedge m \models (l < y) \wedge \\ & \quad \forall (l' < y) \in lits(T) \,.\, m \models \big((l' < y) \implies (l' \leq l)\big) \\ T[-\infty], & \text{otherwise} \end{cases} \tag{7}$$

3.2 Deciding Validity of ∀∃-Formulas

An algorithm AE-VAL for deciding validity of ∀∃-formulas and constructing witnessing Skolem relations was presented in [6]. Without loss of generality, we

Algorithm 1: 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 $S(\vec{x}) \Longrightarrow \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$
   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($\neg T_i$);
```

restrict the input formula to have the form $S(\vec{x}) \implies \exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, where S has no universal quantifiers, and T is quantifier-free.

AE-VAL is an extension of the MBP-algorithm in [13]. It assumes that for each projection T_i there exists a condition ϕ_i under which T is equisatisfiable with T_i:

$$\phi_i(\vec{x}, \vec{y}) \implies \big(T_i(\vec{x}) \iff T(\vec{x}, \vec{y})\big) \tag{8}$$

Intuitively, each ϕ_i captures the substitutions made in T to produce T_i. We assume that each ϕ_i is in the Cartesian form, i.e., a conjunction of terms, in which each $y_j \in \vec{y}$ appears at most once:

$$\phi_i(\vec{x}, \vec{y}) = \bigwedge_{y_j \in \vec{y}} (\psi_{y_j}(\vec{x}, y_j, \ldots, y_n)) \tag{9}$$

Grigory: to re-iterate

We write $(T_i, \phi_i) \leftarrow \textsc{GetMBP}(\vec{y}, m_i, T(\vec{x}, \vec{y}))$ for the invocation of the MBP-algorithm that takes a formula T, a model m_i of T and a vector of variables $\vec{y}$, and returns a projection T_i of T based on m_i and the corresponding relation ϕ_i.

AE-VAL is shown in Alg. 1. Given formulas $S(\vec{x})$ and $\exists \vec{y} \,.\, T(\vec{x}, \vec{y})$, it decides validity of $S(\vec{x}) \implies \exists \vec{y}. T(\vec{x}, \vec{y})$. AE-VAL enumerates the models of $S \wedge T$ and blocks them from S. In each iteration i, it first checks whether S is non-empty (line 3) and then looks for a model m_i of $S \wedge T$ (line 9). If m_i is found, AE-VAL gets a projection T_i of T based on m_i (line 10) and blocks all models contained in T_i from S (line 12). The algorithm iterates until either it finds a model of S that can not be extended to a model of T (line 8), or all models of S are blocked (line 5). In the first case, the input formula is invalid. In the second case, every model of S has been extended to some model of T, and the formula is valid.

```
Algorithm 2: 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 $\pi_{y_j}(\vec{x}, y_j) \leftarrow$ Substitute($\psi_{y_j}(\vec{x}, y_j, y_{j+1}, \ldots, y_n), f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$);
2 if $(\pi_{y_j}(\vec{x}, y_j) \in \varnothing)$ then return $\varnothing$;
3 $\bigwedge_k \left(y_j \sim f_j^k(\vec{x})\right) \leftarrow$ Rewrite($\pi_{y_j}(\vec{x}, y_j)$);
4 return MinimalSkolem$\Big(\bigwedge_k \left(y_j \sim f_j^k(\vec{x})\right)\Big)$;
```

AE-VAL is designed to construct a Skolem relation $Sk_{\vec{y}}(\vec{x}, \vec{y})$, that maps each model of $S(\vec{x})$ to a corresponding model of $T(\vec{x}, \vec{y})$. We use a set of projections $\{T_i(\vec{x})\}$ for $T(\vec{x}, \vec{y})$ and a set of conditions $\{\phi_i(\vec{x}, \vec{y})\}$ that make the corresponding projections equisatisfiable with $T(\vec{x}, \vec{y})$. Intuitively, ϕ_i maps each model of $S \wedge T_i$ to a model of T. Thus, in order to define the guarded Skolem relation $Sk_{\vec{y}}(\vec{x}, \vec{y})$ it is enough to match each ϕ_i against the corresponding T_i:

$$Sk_{\vec{y}}(\vec{x}, \vec{y}) \equiv \begin{cases} \phi_1(\vec{x}, \vec{y}) & \text{if } T_1(\vec{x}) \\ \phi_2(\vec{x}, \vec{y}) & \text{else if } T_2(\vec{x}) \\ \cdots & \qquad \text{else } \cdots \\ \phi_n(\vec{x}, \vec{y}) & \qquad\quad \text{else } T_n(\vec{x}) \end{cases} \tag{10}$$

3.3 Towards Minimal Skolem Refinement

By construction, each local Skolem relation $\phi(\vec{x}, \vec{y})$ has a form $\bigwedge_{y_j \in \vec{y}} (\psi_{y_j}(\vec{x}, y_j, \ldots, y_n))$. Since quantifier elimination in AE-VAL is applied iteratively for each variable $y_j \in \vec{y}$, y_j may depend on the variables of $y_{j+1}, \ldots, y_n$ that are still not eliminated in the current iteration j. Each $\psi_{y_j}(\vec{x}, y_j, \ldots, y_n)$ is the conjunction $\psi_{y_j}(\vec{x}, y_j, \ldots, y_n) = \bigwedge_i (cl_i(\vec{x}, y_j, \ldots, y_n))$, where each cl_i is an (in)equality.

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. 2. 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 $f_{j+1}(\vec{x}), \ldots, f_n(\vec{x})$ (line 1). If for some variable there is no Skolem function to substitute, the algorithm halts with nothing (line 2). Second, the algorithm normalizes $\pi_{y_j}(\vec{x}, y_j)$ into the form $\bigwedge_k \left(y_j \sim f_k(\vec{x})\right)$, i.e., conjunction of expressions, left-hand-sides of which are reserved for y_j and $\sim \in \{<, \leq, =, \geq, >\}$. For this, it uses the method Rewrite (line 3) that rewrites each clause using the following rule (where g, h - are functions over $\vec{x}$, p, q - rational numbers, sgn - a function, returning the sign of the rational number):

$$\Big((g(\vec{x}) + p\times y_j) \sim (h(\vec{x}) + q\times y_j)\Big) \implies \Big((sgn(p-q)\times y_j) \sim (-\frac{g(\vec{x})}{|p-q|}+\frac{h(\vec{x})}{|p-q|})\Big)$$

Finally the algorithm gets rid of inequalities. Method MinimalSkolem rewrites each clause using the following rules:

$$\begin{aligned}
y_j \leq g(\vec{x}) &\implies y_j = g(\vec{x})\\
y_j \geq g(\vec{x}) &\implies y_j = g(\vec{x})\\
y_j < g(\vec{x}) &\implies y_j = g(\vec{x}) - 1\\
y_j > g(\vec{x}) &\implies y_j = g(\vec{x}) + 1\\
y_j > g_1(\vec{x}) \wedge y_j > g_2(\vec{x}) &\implies y_j = max(g_1(\vec{x}), g_2(\vec{x})) + 1\\
y_j < g_1(\vec{x}) \wedge y_j < g_2(\vec{x}) &\implies y_j = min(g_1(\vec{x}), g_2(\vec{x})) - 1\\
y_j > g_1(\vec{x}) \wedge y_j < g_2(\vec{x}) &\implies y_j = \frac{g_2(\vec{x}) - g_1(\vec{x})}{2}\\
&\dots
\end{aligned} \tag{11}$$

4 Case Studies

4.1 A simple controller example

In this section, we provide an illustrative example of how the synthesis algorithm creates a simple implementation from specifications describing the constraints that a controller must meet. The specification is written in the Lustre language, and can be seen in Figure 3. The controller is used to maintain an appropriate level of *speed* at each next step of its execution, using two auxillary signals, called *plus* and *minus* to help determine future decisions on acceleration or deceleration. There is only one input to this example, called *diff*, and is used to compute the amount by which the *speed* value changes with each new state.

The specification is composed of an auxillary node called Sofar, that is a custom operation on a boolean variable to capture whether it has been historically true up to and including the current step. The rest of the nodes defined essentially cover the assumptions and the guarantees that the contract contains. The node *Environment* is used to describe restrictions on the input variable *diff*, while the system's correct response is captured by the node *Property*. The *top* node of the specification is used as the main block of this program, and combines the two constraints to effectively define the structure of the final property that the system should be respecting at all states during its execution.

Andreas: add discription of Controller node. Add text for synthesis case

```
--
-- Source Bertrand Jeannet, NBAC tutorial
--

node Sofar( X : bool ) returns ( Sofar : bool );
let
    Sofar = X -> X and pre Sofar;
tel

node Environment(diff: int; plus,minus: bool) returns (ok: bool);
let
  ok = (-4 <= diff and diff <= 4) and
     (if (true -> pre plus) then diff >= 1 else true) and
     (if (false -> pre minus) then diff <= -1 else true);
tel

node Controller(diff: int) returns (speed: int; plus,minus: bool);
let
  speed = 0 -> pre(speed)+diff;
  plus = speed <= 9;
  minus = speed >= 11;
tel

node Property(speed: int) returns (ok: bool);
var cpt: int;
    acceptable: bool;
let
  acceptable = 8 <= speed and speed <= 12;
  cpt = 0 -> if acceptable then 0 else pre(cpt)+1;
  ok = true -> (pre cpt<=7);
tel

--@ ensures OK;
node top(diff:int) returns (OK: bool);
var speed: int;
    plus,minus,realistic: bool;
let
  (speed,plus,minus) = Controller(diff);
  realistic =  Environment(diff,plus,minus);

  OK = Sofar( realistic and 0 <= speed and speed < 16 ) => Property(speed);
  --%PROPERTY OK;
  --%MAIN;
tel
```

Fig. 3. Specification for a Controller in Lustre

5 Related Work

6 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.: Ae-val: Horn clause-based skolemizer for $\forall\exists$-formulas
6. 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)

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. Heule, M., Seidl, M., Biere, A.: Efficient Extraction of Skolem Functions from QRAT Proofs. In: FMCAD. pp. 107–114. IEEE (2014)
10. John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem Functions for Factored Formulas. In: FMCAD. pp. 73–80. IEEE (2015)
11. Katis, A., Gacek, A., Whalen, M.W.: Machine-checked proofs for realizability checking algorithms (2015), submitted `http://arxiv.org/abs/1502.01292`
12. 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)
13. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. In: Computer Aided Verification. vol. 8559, pp. 17–34. Springer (2014)
14. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT 15(5-6), 455–474 (2013)
15. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36(5), 450–462 (1993)
16. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. pp. 179–190. ACM Press (1989)
17. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS. pp. 404–415. ACM (2006)