Synthesis from Assume-Guarantee Contracts using Inductive Proofs of Realizability

No Author Given

[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] Rockwell Collins Advanced Technology Center
400 Collins Rd. NE, Cedar Rapids, IA, 52498, USA
andrew.gacek@rockwellcollins.com

Abstract. In recent work, we proposed an extention of our contract-based realizability checking algorithm to solve the synthesis problem modulo theories. Given the proof of a contract's realizability we apply a sophisticated skolemizer for $\forall\exists$ formulas to extract an implementation composed of skolem relations that describe the strategies that the implementation needs to follow in order to always comply to the given assume-guarantee contract. The algorithm is similar to k-induction model checking, but involves the use of quantifiers to determine implementability. For the purposes of this paper we developed the proposed synthesis algorithm, and successfully applied it to synthesize implementations for simple case studies. We informally prove the algorithm's correctness and report our results as well as the lessons that we learned during this experiment. Finally, we discuss possible future directions that need to be explored to further improve the algorithm's application domain, optimality and efficiency.

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 [5], 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 [3], using a model checking algorithm that has been formally verified in terms of its soundness for realizable results [4]. Given the inductive proof of realizability, we take advantage of a sophisticated skolemizer for $\forall\exists$ formulas, named AE-VAL [2], 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 3.1 presents our results on using the algorithm to automatically generate leaf-level component implementations for different case studies. Finally, in Section 4 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 5.

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.

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,*

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

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. [1,2] 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 [6] 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 [7], 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:

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

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

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

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.

```
// 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

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 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 Case Studies

3.1 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

4 Related Work

5 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. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Ae-val: Horn clause-based skolemizer for ∀∃-formulas
2. 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)
3. 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)
4. Katis, A., Gacek, A., Whalen, M.W.: Machine-checked proofs for realizability checking algorithms (2015), submitted `http://arxiv.org/abs/1502.01292`
5. 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)

6. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. In: Computer Aided Verification. pp. 17–34. Springer (2014)
7. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal 36(5), 450–462 (1993)