Towards Synthesis from Assume-Guarantee Contracts involving Infinite Theories: A Preliminary Report

Andreas Katis
Department of Computer Science and Engineering
University of Minnesota
200 Union Street
Minneapolis, MN 55455,USA
katis001@umn.edu

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

Michael W. Whalen
Department of Computer Science and Engineering
University of Minnesota
200 Union Street
Minneapolis, MN 55455,USA
whalen@cs.umn.edu

ABSTRACT

In previous work, we have introduced a contract-based *realizability checking* algorithm for assume-guarantee contracts involving infinite theories, such as linear integer/real arithmetic and uninterpreted functions over infinite domains. This algorithm can determine whether or not it is possible to construct a realization (i.e. an implementation) of an assume-guarantee contract. The algorithm is similar to k-induction model checking, but involves the use of quantifiers to determine implementability.

While our work on realizability is inherently useful for *virtual integration* in determining whether it is possible for suppliers to build software that meets a contract, it also provides the foundations to solving the more challenging problem of component synthesis. In this paper, we provide an initial synthesis algorithm for assume-guarantee contracts involving infinite theories. To do so, we take advantage of our realizability checking procedure and a skolemization solver for ∀∃-formulas, called AE-VAL. We show that it is possible to immediately adapt our existing algorithm towards synthesis by using this solver, using a demonstration example. We then discuss challenges towards creating a more robust synthesis algorithm.

1. INTRODUCTION

The problem of automated synthesis of reactive systems using from propositional specifications is a very well studied area of research [1]. By definition, the problem of synthesis entails the discovery of efficient algorithms able to construct a candidate program that is guaranteed to comply with the predefined specification. Inevitably, the related work on synthesis has tackled several sub-problems, such as that of function and template synthesis, as well as the weaker problem regarding the implementability, or otherwise, realizability of the specification.

In a similar fashion, a collaboration between Rockwell Collins and the University of Minnesota has focused on designing tools that provide compositional proofs of correctness [2, 3, 4, 5]. In the context of synthesis, we recently introduced a decision procedure for determine the *realizability* of contracts involving infinite theories such as linear integer/real arithmetic and/or uninterpreted functions that is checkable by any SMT solver that supports quantification [6]. Furthermore, in [7] we formally proved the soundness of our checking algorithm using the Coq interactive theorem prover. The realizability checking procedure is now part of the AGREE reasoning framework [2], which supports compositional assume-guarantee contract reasoning over system architectural models written in AADL [8].

While checking the realizability of contracts provided us with fruitful results and insight in several case studies, it also worked as solid ground towards the development of an automatic component synthesis procedure. The most important obstacle initially, was the inability of the SMT solver to handle higher-order quantification. Fortunately, interesting directions to solving this problem have already surfaced, either by extending an SMT solver with native synthesis capabilities[9], or by providing external algorithms that reduce the problem by efficient quantifier elimination methods [10].

The main contribution of this paper is the implementation of a component synthesis algorithm for infinite theories, using specifications expressed in assume-guarantee contracts. The algorithm heavily relies on our previous implementation for realizability checking, but also takes advantage of a recently published skolemizer for ∀∃-formulas, named AE-VAL. The main idea in this implementation is to effectively extract a Skolem relation that is essentially, a collection of strategies, that can directly lead to an implementation which is guaranteed to comply to the corresponding contract.

In Section 2 we provide the necessary background definitions from our previous work on realizability checking. Section 3 presents our approach to solving the synthesis problem for assume-guarantee contracts using theories. Finally, in Section 4 we give a brief historical background on the related research work on synthesis, and we report our conclusions and upcoming future work in Section 5.

2. PRELIMINARIES

In the remainder of the paper, we endeavor to solve a synthesis problem for assume-guarantee contracts involving infinite theories. Formally, an implementation is a set of valid initial states I and transition relation T that implements the

FormaliSE'16, May 15 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897667.2897675

contract. In this section, we introduce the necessary formal machinery to talk about *realizations* of an assume-guarantee contract.

2.1 Example

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.

Figure 1: Example of a realizable contract

One can easily prove that an implementation able to satisfy the contract is possible. The same though does not apply in the case where we omit the assumption from our original contract. Given no constraints over the values that the inputs can take, we have a case where the implementation may behave in an inconsistent manner regarding the value of z, thus making the violation of the guarantees possible. In the first case, the contract is realizable, but in the second, we cannot find an implementation that can provide us with an output that satisfies the contract guarantees, for any valid input. These contracts are considered to be unrealizable.

2.2 Formal Definitions

We use the types *state* and *inputs* to describe a state and the set of inputs in the system, respectively. We define a *transition system* as a pair (I, T), where I is the set of initial states, of type *state* $\rightarrow$ *bool* and T is the transition relation, of type *state* $\rightarrow$ *inputs* $\rightarrow$ *state* $\rightarrow$ *bool*.

A contract in this context is defined by its *assumptions* and *guarantees*. The *assumptions* A impose constraints over the inputs, while the *guarantees* are further decomposed into the pair (G_I, G_T) with G_I describing the valid initial states for the system, and G_T specifying the new states to which the system may transition, given a specific state and input. 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.

Given the above, we expressed realizability as follows. A state s is *viable* if, starting from s, the transition system is capable of continuously responding to incoming valid inputs. Alternatively, s is *viable* if the transitional guarantee G_T infinitely holds, given valid inputs. As such, viability is defined coinductively:

$$Viable(s) = \forall i. A(s,i) \Rightarrow \exists s'. G_T(s,i,s') \wedge Viable(s') \quad (1)$$

Using the definition of *viable*, a contract is *realizable* if and only if

$$\exists s.\ G_I(s) \wedge Viable(s). \quad (2)$$

To use this coinductive formula in a model checking framework, we had to further massage the definition into one that resembles the principle of *k-induction*. As such, the base step of the induction ensures that given a state, G_T can keep responding to valid inputs for at least n steps in the future. We call this state *finitely viable*, written $Viable_n(s)$:

$$\begin{aligned} &\forall i_1. A(s,i_1) \Rightarrow \exists s_1. G_T(s,i_1,s_1) \wedge \\ &\quad \forall i_2. A(s_1,i_2) \Rightarrow \exists s_2. G_T(s_1,i_2,s_2) \wedge \ldots \wedge \\ &\quad\quad \forall i_n. A(s_{n-1},i_n) \Rightarrow \exists s_n. G_T(s_{n-1},i_n,s_n) \end{aligned} \quad (3)$$

On the other hand, the inductive step checks whether a path starting from a finitely viable state can be further extended one-step. We call states that build such paths *extendable*, written $Extend_n(s)$:

$$\begin{aligned} &\forall i_1, s_1, \ldots, i_n, s_n. \\ &\quad A(s,i_1) \wedge G_T(s,i_1,s_1) \wedge \ldots \wedge \\ &\quad\quad A(s_{n-1},i_n) \wedge G_T(s_{n-1},i_n,s_n) \Rightarrow \\ &\quad\quad\quad \forall i. A(s_n,i) \Rightarrow \exists s'. G_T(s_n,i,s') \end{aligned} \quad (4)$$

Considering these underapproximations, the algorithm is split into two separate checks. For the *BaseCheck*, we ensure that there exists an initial finitely viable state,

$$BaseCheck(n) = \exists s. G_I(s) \wedge Viable_n(s) \quad (5)$$

while the *ExtendCheck* tries to prove that all valid states are extendable.

$$ExtendCheck(n) = \forall s. Extend_n(s) \quad (6)$$

Due to the definition of finite viability containing $2n$ quantifier alternations, we can not practically use *BaseCheck* as the current SMT solvers struggle solving formulas of such structure. Therefore we finally proposed a simplified version of *BaseCheck*, which essentially tries to prove that all initial states are extendable for any $k \leq n$.

$$BaseCheck'(n) = \forall k \leq n. (\forall s. G_I(s) \Rightarrow Extend_k(s)) \quad (7)$$

Even though the simplified definition is more simple for an SMT solver to process, it comes with a cost, as it introduces cases of realizable contracts that are considered to be unrealizable from the algorithm. To the extent of our experiments, such a case has yet to be met, as it inherently requires the user to purposely define contracts of such behavior.

3. SYNTHESIS FROM CONTRACTS

With a sound implementation of the realizability checking algorithm at hand, the next step was to tackle the more interesting problem of synthesis, i.e. the automated derivation of implementations that would be safe in terms of satisfying the constraints defined by the component's contract. The intuition behind solving the synthesis problem in our context relies on finding a set of initial states I and a transition relation T that would satisfy the requirements specified in the contract. Unfortunately, the lack of power in SMT solvers in terms of solving formulas that contain higher-order quantification immediately ruled out the prospect of using one as our primary synthesis tool. Therefore, an alternative work from Fedyukovich et al. [10, 11] on a skolemizer for $\forall\exists$-formulas on linear integer arithmetic was chosen to be used as a means of extracting a witness that could directly be used in component synthesis.

The tool, called AE-VAL is using the Model-Based Projection technique in [12] to validate $\forall\exists$-formulas, based on Loos-Weispfenning quantifier elimination [13]. As part of the procedure, a Skolem relation is provided for the existentially quantified variables 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 simple. Consider the checks 7 and 6 that the realizability checking algorithm is using. *BaseCheck'* is still necessary for the synthesis problem to ensure that all initial states in the problem are valid. *ExtendCheck* on the other hand can be further used in actually synthesizing implementations. The check tries to prove that every valid state in our system is extendable, i.e. all states can be starting points to paths that comply to the system contract, and furthermore are extendable by one step:

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

which can be rewritten:

$$\begin{aligned}&\forall i_1, s_1, \ldots, i_n, s_n, i.\\ &\quad A(s, i_1) \wedge G_T(s, i_1, s_1) \wedge \ldots \wedge\\ &\quad A(s_{n-1}, i_n) \wedge G_T(s_{n-1}, i_n, s_n) \wedge A(s_n, i) \Rightarrow\\ &\qquad\qquad \exists s'. G_T(s_n, i, s')\end{aligned} \quad (8)$$

Such a formula is exactly what is required by a $\forall\exists$ solver such as AE-VAL in order to produce the witness for the existential variables s'. AE-VAL solves this formula by providing an assignment for each existential variable in a piecewise relation based on a partitioning based on assignments to the universal variables. In other words, by examining a bounded history of the state and input variable values in the contract (the universally quantified variables in Formula 8), we determine the next values of the state variables. An example of a portion of this partitioning is shown in Figure 3. In other words, the Skolem relation contains, starting from a valid initial state of variables, strategies in terms of how the new state is selected, in such a way, that the contract is not violated.

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 solver 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

```
// 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;
}
```

Figure 2: Algorithm skeleton for synthesis

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.

3.1 Synthesis Example

As an example that demonstrates the process, consider the contract created for a mode controller in a simple microwave model of 260 lines of code that was used as one of the base case studies in [6]. The controller has four inputs, *start* which is used to indicate whether the microwave is at an initial state or not, *clear* that is used as a stop signal for the system, *seconds_to_cook* as a countdown timer and *door_closed* as an indicator that the microwave's door is closed or not. The controller returns the current state of the microwave's mode using *cooking_mode*. The contract consists of one assumption and nine guarantees, which are shown below informally, as well as formally in AADL. A library named *defs* is used to define auxiliary functions, such as *rising_edge()* which returns "true" when the corresponding signal is at its rising edge, and *initially_true()* which is used to check a variable's value at the component's initial state.

MC Assumption – seconds_to_cook is greater than or equal to zero.
`seconds_to_cook >= 0;`

MC Guarantee-0 – The range of the cooking_mode variable shall be [1..3].
`cooking_mode >= 1 and cooking_mode <= 3;`

MC Guarantee-1 – The microwave shall be in cooking mode only when the door is closed.
`is_running => door_closed;`

MC Guarantee-2 – The microwave shall be in setup mode in the initial state.
`(defs.initially_true(start)) => is_setup;`

MC Guarantee-3 – At the instant the microwave starts running, it shall be in the cooking mode if the door is closed.

```
(defs.rising_edge(is_running) and
               door_closed) => is_cooking;
```

MC Guarantee-4 – At the instant the microwave starts running, it shall enter the suspended mode if the door is open.

```
(defs.rising_edge(is_running) and
        not door_closed) => is_suspended;
```

MC Guarantee-5 – At the instant the clear button is pressed, if the microwave was cooking, then the microwave shall stop cooking.

```
(defs.rising_edge(clear) and
        is_cooking) => not is_cooking;
```

MC Guarantee-6 – At the instant when the clear button is pressed, if the microwave is in suspended mode, it shall enter the setup mode.

```
(defs.rising_edge(clear) and
        is_suspended) => is_setup;
```

MC Guarantee-7 – If suspended, at the instant the start key is pressed the microwave shall enter cooking mode if the door is closed.

```
(defs.rising_edge(start) and is_suspended
        and door_closed) => is_cooking;
```

MC Guarantee-8 – If seconds_to_cook = 0, microwave will be in setup mode.

```
(seconds_to_cook = 0) => is_setup;
```

The contract is then translated from AADL into an equivalent Lustre program that is then given as input to the JKind model checker [14], where our realizability algorithm is implemented as a separate feature. From JKind, the Lustre specification is further translated into the SMT-LIB v2 format. In reality, the program is split into two different processes that run in parallel and correspond to the checks 6 and 7 that our synthesis algorithm is using. Considering the fact that the contract is realizable, we impose the negation of our target ∀∃-formula as a query to the AE-VAL skolemizer during the last step of *ExtendCheck*. AE-VAL responds that the original formula can be satisfied, and provides a Skolem relation, a part of which is shown in Figure 3.

As seen in Figure 3, the Skolem relation is composed of nested *if-then-else* blocks, which indicate the possible valid transitions the implementation can follow given a specific state, without violating the contract. Each variable appearing in the conjuncts of the relation is named uniquely after the state that it refers to, using the *$X* postfix, where X is an integer. In addition to each state's variables, we keep track of whether each state is initial using the variable *%init*.

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

```
ite([&&
    $defs__rising_edge~1.Mode_Control_Impl_Instance__signal$0
    !($Mode_Control_Impl_Instance__seconds_to_cook$0>=0)
    !$defs__initially_true~0.Mode_Control_Impl_Instance__result$0
  ], [&&
    $Mode_Control_Impl_Instance__is_setup$0
    $defs__rising_edge~1.Mode_Control_Impl_Instance__re$0
    !$Mode_Control_Impl_Instance__is_cooking$0
    $defs__rising_edge~1.Mode_Control_Impl_Instance__signal$0
    !$_TOTAL_COMP_HIST$0
    !$_SYSTEM_ASSUMP_HIST$0
    !$Mode_Control_Impl_Instance__is_suspended$0
    !$Mode_Control_Impl_Instance__is_running$0
    !$defs__rising_edge~0.Mode_Control_Impl_Instance__re$0
    !$defs__initially_true~0.Mode_Control_Impl_Instance__b$0
    !$defs__initially_true~0.Mode_Control_Impl_Instance__result$0
    !$defs__rising_edge~2.Mode_Control_Impl_Instance__re$0
    !$defs__rising_edge~2.Mode_Control_Impl_Instance__signal$0
  ], ite([&&
      %init
      $_SYS_GUARANTEE_2$0
      !($Mode_Control_Impl_Instance__seconds_to_cook$0>=0)
      !$defs__rising_edge~1.Mode_Control_Impl_Instance__signal$0
      !$defs__initially_true~0.Mode_Control_Impl_Instance__b$0
    ], ...))
```

Figure 3: A portion of the Skolem relation generated for the Microwave Mode Logic

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.

4. RELATED WORK

The problem of program synthesis was first expressed formally in the early 1970s [15] as a potentially important area of study and research. Pnuelli and Rosner use the term *implementability* in [16] to refer to the problem of synthesis for propositional LTL. Additionally, the authors in [16] proved that the lower-bound time complexity of the problem is doubly exponential, in the worst case. In the following years, several techniques were introduced to deal with the synthesis problem in a more efficient way for subsets of propositional LTL [17], simple LTL formulas ([18], [19]), as well as in a component-based approach [20] and specifications based on other temporal logics ([21], [22]), such as SIS [23].

In 2010, a survey from Sumit Gulwani described the directions that future research will focus on, towards the road of fully automated synthesis of programs [1]. The approaches that have been proposed are many, and differ on many aspects, either in terms of the specifications that are being exercised, or the reasoning behind the synthesis algorithm itself. On the one hand, template-based synthesis [24] is focused on the exploration of programs that satisfy a specification that is refined after each iteration, following the basic principles of deductive synthesis. Inductive synthesis, on the other hand, is an active area of research where the main goal is the generation of an inductive invariant that can be used to describe the space of programs that are guaranteed to satisfy the given specification [25]. This idea is mainly supported by the use of SMT solvers to guide the invariant refinement through traces that violate the requirements, known as counterexamples. Recently published work on extending SMT solvers with counterexample-guided synthesis shows that they can eventually be used as an alternative to solving the problem under certain domains of arithmetic [9].

Finally, an interesting and relevant work has been done regarding the solution to the controllability problem using in

[26] [27] and [28], which involves the decision on the existence a strategy that assigns certain values to a set of controllable activities, with respect to a set of uncontrollable ones.

Our approach relies on the idea of extracting programs that satisfy the constraints from the proof of their realizability that is produced by a sophisticated theorem prover. The proof itself is provided through a model checking approach that follows the k-induction principle. To the best of our knowledge this is the first attempt on providing a synthesis algorithm for an assume-guarantee framework, using infinite theories.

5. CONCLUSION

In this paper we present the first known algorithm of implementation synthesis from assume-guarantee contracts, using theories. To achieve this, we took advantage of our previous work, by extracting programs directly from the contract's proof of realizability. Additionally, the algorithm depends on the extraction of Skolem relations from the AE-VAL decision procedure for $\forall\exists$-formulas.

Future work involves exploring the solution to many obstacles that stand still. First, we want to aim towards extending our current approach to other theories like linear real arithmetic, as AE-VAL currently only supports integer arithmetic. Another goal that we are interested in exploring is the definition of a better realizability checking algorithm based on the idea of invariant generation, using the idea of property directed reachability [29, 30, 31]. Another problem to potentially consider are cases where the provided implementation cannot actually be used in practice. This is an interesting area of research due to the use of infinite theories in our approach, which may result in implementations that use infinite precision, a feature that cannot be practically achieved by any real program.

Several other directions to improving our existing synthesis algorithm involve the improvement of representations in our context. For example the transition relation often takes up a big portion of the final SMT-LIB output that is given to AE-VAL to process, and is relatively hard to process. The same applies to the Skolem relation, which for this example is almost 900 lines of nested *if-then-else* blocks. An interesting approach to improving the algorithm's performance relies in the translation of the original data-flow program from Lustre to a finite state machine, using a sophisticated compilation methods as the ones presented in [32]. The disadvantage of using this approach is mainly that the final state machine is not guaranteed to be minimal, due to the declarative nature of the programs that we exercise. As a final remark, we intend to formally verify the synthesis algorithm presented in this paper, by extending the proof that has already been constructed for our algorithm on realizability checking.

6. 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).

7. REFERENCES

[1] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming.* ACM, 2010, pp. 13–24.

[2] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, A. E. Goodloe and S. Person, Eds., vol. 7226. Berlin, Heidelberg: Springer-Verlag, April 2012, pp. 126–140.

[3] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. Heimdahl, and S. Rayadurgam, "Your what is my how: Iteration and hierarchy in system design," *Software, IEEE*, vol. 30, no. 2, pp. 54–60, 2013.

[4] A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl, "Compositional verification of a medical device system," in *ACM Int'l Conf. on High Integrity Language Technology (HILT) 2013.* ACM, November 2013.

[5] J. Backes, D. Cofer, S. Miller, and M. W. Whalen, "Requirements analysis of a quad-redundant flight control system," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, K. Havelund, G. Holzmann, and R. Joshi, Eds., vol. 9058. Springer International Publishing, 2015, pp. 82–96. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-17524-9_7

[6] A. Gacek, A. Katis, M. W. Whalen, J. Backes, and D. Cofer, "Towards realizability checking of contracts using theories," in *NASA Formal Methods.* Springer, 2015, pp. 173–187.

[7] A. Katis, A. Gacek, and M. W. Whalen, "Machine-checked proofs for realizability checking algorithms," 2015, submitted http://arxiv.org/abs/1502.01292.

[8] SAE-AS5506, "Architecture analysis and design language," Nov 2004.

[9] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, "Counterexample-guided quantifier instantiation for synthesis in smt."

[10] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Ae-val: Horn clause-based skolemizer for $\forall\exists$-formulas."

[11] ——, "Automated discovery of simulation between programs," *submitted, also available as Technical Report USI*, vol. 5, 2014.

[12] A. Komuravelli, A. Gurfinkel, and S. Chaki, "Smt-based model checking for recursive programs," in *Computer Aided Verification.* Springer, 2014, pp. 17–34.

[13] R. Loos and V. Weispfenning, "Applying linear quantifier elimination," *The Computer Journal*, vol. 36, no. 5, pp. 450–462, 1993.

[14] A. Gacek, "JKind – an infinite-state model checker for safety properties in Lustre," http://loonwerks.com/tools/jkind.html, 2016.

[15] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, pp. 151–165, 1971.

[16] A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," *Proceedings of the 16th ACM*

SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'89), pp. 179–190, 1989.

[17] U. Klein and A. Pnueli, "Revisiting Synthesis of GR(1) Specifications," *Proceedings of the 6th International Conference on Hardware and Software: Verification and Testing (HVC'10)*, pp. 161–181, 2010.

[18] A. Bohy, V. BruyÃÍre, E. Filiot, N. Jin, and J.-F. Raskin, "Acacia+, a tool for LTL Synthesis," *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*, pp. 652–657, 2012.

[19] S. Tini and A. Maggiolo-Schettini, "Compositional Synthesis of Generalized Mealy Machines," *Fundamenta Informaticae*, vol. 60, no. 1-4, pp. 367–382, 2003.

[20] K. Chatterjee and T. A. Henzinger, "Assume-Guarantee Synthesis," *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, pp. 261–275, 2007.

[21] N. Beneš, I. Černá, and F. Štefaňák, "Factorization for component-interaction automata," in *SOFSEM 2012: Theory and Practice of Computer Science.* Springer, 2012, pp. 554–565.

[22] J. Hamza, B. Jobstmann, and V. Kuncak, "Synthesis for Regular Specifications over Unbounded Domains," *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pp. 101–109, 2010.

[23] A. Aziz, F. Balarin, R. Braton, and A. Sangiovanni-Vincentelli, "Sequential Synthesis using SIS," *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'95)*, pp. 612–617, 1995.

[24] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 497–518, 2013.

[25] P. Flener and D. Partridge, "Inductive programming," *Automated Software Engineering*, vol. 8, no. 2, pp. 131–137, 2001.

[26] A. Cimatti, A. Micheli, and M. Roveri, "Solving temporal problems using SMT: Weak controllability," in *AAAI*, 2012, pp. 448–454.

[27] ——, "Solving temporal problems using SMT: Strong controllability," in *CP*, 2012, pp. 248–264.

[28] ——, "Solving strong controllability of temporal problems with uncertainty using SMT," *Constraints*, 2014.

[29] A. Bradley, "SAT-based model checking without unrolling," *VMCAI*, 2011.

[30] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Ic3 modulo theories via implicit predicate abstraction," in *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2014, pp. 46–61.

[31] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Formal Methods in Computer-Aided Design (FMCAD), 2011.* IEEE, 2011, pp. 125–134.

[32] N. Halbwachs, P. Raymond, and C. Ratel, "Generating efficient code from data-flow programs," in *Third Int'l Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.