Interpolation for Synthesis on Unbounded Domains

Viktor Kuncak and Régis Blanc
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

***Abstract*—Synthesis procedures compile relational specifications into functions. In addition to bounded domains, synthesis procedures are applicable to domains such as mathematical integers, where the domain and range of relations and synthesized code is unbounded. Previous work presented synthesis procedures that generate self-contained code and do not require components as inputs. The advantage of this approach is that it requires only specifications as user input. On the other hand, in some cases it can be desirable to require that the synthesized system reuses existing components. This paper describes a technique to automatically synthesize systems from components. It is also applicable to repair scenarios where the desired sub-component of the system should be replaced to satisfy the overall specification. The technique is sound, and it is complete for constraints for which an interpolation procedure exists, which includes e.g. propositional logic, bitvectors, linear integer arithmetic, recursive structures, finite sets, and extensions of the theory of arrays.**

I. Introduction

Software synthesis is an active area of research [5], [13], [14] and has a long tradition [1], [10], [12]. We here pursue synthesis of functions from inputs to outputs that are guaranteed to satisfy a given input/output relation expressed in a decidable logic. Such approach have been referred to as complete functional synthesis [7], [8]. The appeal of this direction is that it synthesizes functions over unbounded domains whenever they exist, and that the produced code is guaranteed to satisfy the specification for the entire unbounded range of inputs. Synthesis procedures for propositional logic, linear rational arithmetic, and Boolean Algebra with Presburger Arithmetic and parametrized coefficients are presented in [6]–[8]. Synthesis procedures for algebraic data types and arrays are presented in [2].

The previous work demonstrated synthesis procedures that generate self-contained code and do not require components as inputs. This approach requires only the input/output specification as the user input. This is in contrast to some of the existing approaches that require components as inputs and enumerate different combinations of the components, checking which ones satisfy a specification. In general, however, synthesis from components is not only a way to simplify the synthesis task, but also a way to control the outcome of synthesis, making the process more predictable. It can be desirable to require synthesis procedures to reuse existing functionality, even if there exists a method to synthesize the system from scratch. For example, using existing components may have expected cost metrics in terms of computational complexity, or market availability. This paper presents techniques that can be used to ensure that a synthesis procedure reuses a given set of components in the synthesized code. The work in reactive LTL synthesis from components [9] deals with stateful reactive components but is limited to finite-state systems and encounters a 2EXPTIME lower bound, whereas we work in the stateless scenario but for infinite domains where we can leverage modern SMT solvers.

Our inspiration comes from generalizing methods such as resynthesis, which have proved useful for generation of combinational circuits [4], [15]. These techniques perform case analysis on boolean variables in the output, which makes them specific to finite domains. We show, however, that such complete technique can be devised for every decidable domain for which interpolation and synthesis procedures exists. This includes bitvector domains, potentially allowing synthesis of circuits at a higher level, as well as the domain of structures used in software, such as recursive algebraic data types, sets, linear integer arithmetic, and arrays. For the approach to work in practice, what is needed are well-behaved interpolation procedures that prefer simpler and computationally shorter interpolants, a requirement that is in any case desirable for interpolation in predicate abstraction refinement [3].

II. Background: Synthesis as Relation Transformation

The starting point for our work is the framework for functional synthesis, as presented most recently in [2], whose notation we follow. For a high-level overview, please consult [7]. A *synthesis problem* is a triple $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$, where $\bar{a}$ is a set of *input* variables, $\bar{x}$ is a set of *output* variables and ϕ is a formula whose free variables are a subset of $\bar{a} \cup \bar{x}$. A synthesis problem denotes a binary relation $\{(\bar{a}, \bar{x}) \mid \phi\}$ between inputs and outputs. The goal of synthesis is to transform such relations until they become executable programs. Programs correspond to formulas of the form $P \wedge (\bar{x} = \bar{T})$ where $\mathrm{vars}(P) \cup \mathrm{vars}(\bar{T}) \subseteq \bar{a}$. We denote programs by $\langle P \mid \bar{T}\rangle$. We call the formula P a *precondition* and call the term $\bar{T}$ a *program term*. We use $\vdash$ to denote the transformation on synthesis problems, so

$$[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash [\![\bar{a}\ \langle\phi'\rangle\ \bar{x}]\!] \tag{1}$$

means that the problem $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$ can be transformed into the problem $[\![\bar{a}\ \langle\phi'\rangle\ \bar{x}]\!]$. The variables on the right-hand side are always the same as on the left-hand side. Our goal is to compute, given $\bar{a}$, one value of $\bar{x}$ that satisfies ϕ. We therefore define the soundness of (1) as a process that refines the binary relation given by ϕ into a smaller relation given by ϕ', without reducing its domain. Expressed in terms of

formulas, the conditions become the following:

$$\begin{array}{rcll} \phi' & \models & \phi & \textit{refinement} \\ \exists \bar{x}.\phi & \models & \exists \bar{x}.\phi' & \textit{domain preservation} \end{array}$$

In other words, $\vdash$ denotes domain-preserving refinements of relations. Note that the dual entailment $\exists \bar{x}.\phi' \models \exists \bar{x}.\phi$ also holds, but it follows from *refinement*. Note as well that $\vdash$ is transitive. In most cases we will consider transformations whose result is a program: $[\![\bar{a}\ \langle \phi \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle$. The correctness of such transformations reduces to

$$\begin{array}{rcll} P & \models & \phi[\bar{x} \mapsto \bar{T}] & \textit{refinement} \\ \exists \bar{x}.\phi & \models & P & \textit{domain preservation} \end{array}$$

A *synthesis procedure* for a theory $\mathcal{T}$ is given by a set of inference rules and a strategy for applying them such that every formula in the theory is transformed into a program.

III. Interpolation for Synthesis from Components

We next show how synthesis procedure, even for unbounded domains, can leverage interpolation techniques to synthesize a function as a combination of other functions. This enables the user to control the synthesis process by requiring that the desired function is realized as a combination of results of given functions. The technique presented here is inspired by asking whether the finite-state resynthesis techniques from [4], which was experimentally shown to be useful in practice, could be lifted from propositional to the level of first-order theories. The key difficulty is that case analysis on the output, performed in [4], is not possible for infinite-domain theories. We present instead a more general formulation, which works in two stages: 1) construct a quantifier-free input/output constraint describing the implementation of the desired functionality from components, using interpolation for the theory of interest; and 2) synthesize the implementation from the input/output constraint, using the appropriate synthesis procedure.

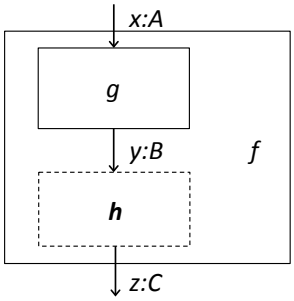
A. Synthesis from Components as a Two-Step Process

Figure 1 summarizes the rules for synthesis from components. The general setup is given by the rule 'COMP' in Figure 1, which is a simple fact of first-order logic with equality. Given a function $f : A \to C$, we encode the available components as another function $g : A \to B$. Note that B can be a cross-product of any number of simpler domains, so g can encode any finite number of component functions. The goal is to express f in terms of the result of g. In other words, we seek a function h such that $f(x) = h(g(x))$. 'COMP' rule gives one way to find such function h:

1) construct a relational description I of the desired h; we say I is a *relational connector* for obtaining f from g.
2) find h as a refinement of the relational connector I.

B. Correctness of Synthesis from a Relational Connector

To see why 'COMP' is correct, let the two assumptions in the rule hold, let x be arbitrary, define z by $z = f(x)$ and b by $b = g(x)$. Then $I(g(x), z)$ by the first assumption, so $I(g(x), h(g(x)))$ by the second assumption. Using the first assumption once again (with $h(g(x))$ as an instance of the universally quantified z), we conclude $h(g(x)) = f(x)$, as desired.



$$\frac{\begin{array}{c} f : A \to C \\ g : A \to B \qquad h : B \to C \qquad I \subseteq B \times C \\ \forall x, z.\ I(g(x), z) \leftrightarrow z = f(x) \\ \forall b.\ (\exists z. I(b, z)) \to I(b, h(b)) \end{array}}{\forall x. f(x) = h(g(x))}\ \text{COMP}$$

$$\frac{\begin{array}{c} f(x_1) = z_1 \wedge g(x_1) = y_1 \wedge y_1 = y_2 \models I(y_2, z_1) \\ I(y_2, z_1) \models (g(x_2) = y_2 \wedge f(x_2) = z_2 \to z_1 = z_2) \end{array}}{\boxed{\forall x, z.\ I(g(x), z) \leftrightarrow z = f(x)}}\ \text{INT-UNIQ}$$

$$\frac{\begin{array}{c} \text{vars}(I) \subseteq \{b, z\} \\ \boxed{\forall x, z.\ I[b := g(x)] \leftrightarrow z = f(x)} \qquad [\![b\ \langle I \rangle\ z]\!] \vdash \langle P \mid H \rangle \end{array}}{[\![x\ \langle z = f(x) \rangle\ z]\!] \vdash \langle \top \mid H[b := g(x)] \rangle}\ \text{COMP-S}$$

Fig. 1. Synthesis from Components

C. Finding Relational Connector Using Interpolation

We next turn to the problem of finding the relational connector I. The key insight is that a single call to a theorem prover that can compute interpolants [11] is sufficient to find I with the desired property, $\forall x, z.\ I(g(x), z) \leftrightarrow z = f(x)$. This is captured by the 'INT-UNIQ' rule, which stands for "interpolating uniqueness".

To understand the rule, observe that it contains two entailments (universally quantified implications), which, chained together, can be represented as the following property of f and g:

$$\frac{f(x_1) = z_1 \wedge g(x_1) = y_1 \wedge y_1 = y_2}{g(x_2) = y_2 \wedge f(x_2) = z_2 \to z_1 = z_2}$$

By rearranging the order of assumptions, we can equivalently write this condition as:

$$\frac{\begin{array}{c} g(x_1) = y_1 \qquad g(x_2) = y_2 \qquad y_1 = y_2 \\ f(x_1) = z_1 \qquad f(x_2) = z_2 \end{array}}{z_1 = z_2} \tag{2}$$

This condition states that if g computes the same result on two arguments x_1, x_2, then so does f. Such condition is necessary for the existence of a function h that would enable us to compute $f(x)$ as $h(g(x))$. Indeed, if $g(x_1) = g(x_2)$ then

$h(g(x_1)) = h(g(x_2))$, so we need to have also $f(x_1) = f(x_2)$. Therefore, whenever we can hope to find a function h, we know that the above implication holds. Moreover, if the logic in which f, g are described has the interpolation property, we know that an interpolant I exists. For a decidable logic with interpolation property, rule 'INT-UNIQ' gives an effective algorithm for computing I from f and g.

D. Why Interpolants Precisely Characterize Relational Connectors

We have seen that an I can be found such that the assumptions of the 'INT-UNIQ' rule hold. This ensures that the assumptions of 'INT-UNIQ' rule can be satisfied in practice. We next show the correctness of 'INT-UNIQ': any I that is found in such interpolation process satisfies the conclusion of the 'INT-UNIQ' rule, so it can be used in the 'COMP' rule. Consider the first assumption of 'INT-UNIQ':

$$f(x_1) = z_1 \wedge g(x_1) = y_1 \wedge y_1 = y_2 \models I(y_2, z_1)$$

Using one-point rule we eliminate y_1 and y_2, replacing them with $g(x_1)$. The result is

$$f(x_1) = z_1 \models I(g(x_1), z_1) \tag{3}$$

Consider the second rule:

$$I(y_2, z_1) \models (g(x_2) = y_2 \wedge f(x_2) = z_2 \rightarrow z_1 = z_2)$$

Using one-point rule we replace y_2 with $g(x_2)$ and replace z_2 with $f(x_2)$, obtaining

$$I(g(x_2), z_1) \models z_1 = f(x_2) \tag{4}$$

By renaming the variables and conjoining (3) and (4), we obtain the desired equivalence:

$$\forall x, y.\ I(g(x), z) \leftrightarrow z = f(x)$$

E. Informal Summary of the Idea

In summary, to express $f(x)$ as $h(g(x))$, we state a necessary condition (2) for f to depend only on the result of g, writing it in a flat form. We then split conjuncts in such a way to separate two uses if f, g between the two sides of the interpolant. Such split leads to interpolants that precisely specify the relationship between the variables y and z, which is the relationship I that we wish to synthesize.

F. From Relation to Function Using Synthesis Procedures

The relational connector I is a relation, so we wish to find a function that refines it. This is where the idea of synthesis using interpolation connects to the framework of synthesis procedures [2], [6]–[8]. The result is a syntactic variant of the rule 'COMP', which we denote 'COMP-S' in Figure 1. The relational connector I is now represented as a formula I with free variables: b (ranging over the set B, the results of g) and z (the desired result of the computation of f and h). Application that was expressed in 'COMP' as $I(g(x), z)$ therefore becomes the substitution $I[b := g(x)]$. Similarly, the desired function h is expressed as a syntactic term H with the free variable b. The condition $f(x) = h(g(x))$ then becomes a synthesis step that transforms $f(x)$ into the term $H[b := g(x)]$ that has x as the free variable.

The key step is $[\![b\ \langle I \rangle\ z]\!] \vdash \langle P \mid H \rangle$, which takes the relational connector and transforms it into a function given by H. In the process, it generates the most general precondition, P. In terms of the rule 'COMP', the condition P corresponds to the condition $\exists z. I(b, z)$, because of the domain-preservation requirement of the "$\vdash$" operator.

Intuitively, because I is only applied to values $g(x)$, the precondition P contains the range of g, so it becomes trivially satisfied in the overall function $h(g(x))$. This allows us to synthesize a program $H[b := g(x)]$ with a trivial, true, precondition $\top$ in the conclusion of the rule.

IV. Further Generalizations

Note that our results apply to any theory for which we have synthesis procedures. The discovery of a relational connector does not even require a synthesis procedure, only interpolation. In practice, we have demonstrated synthesis for many theories and they typically have interpolation [2], [7], [8].

A. Tuples and Passing Inputs

Recall that in the original problem we synthesize $h(y)$ such that $f(x)$ is $h(g(x))$. Note that adding the notion of n-tuples does not change decidability in most cases, because tuple variables can be replaced by individual variables. Thus, we may assume, when convenient, that x is a vector and that g returns a vector.

It can be useful to make some of the coordinates of x directly available to h. To describe this case, we let $x = (x_0, x_1)$ and let $g(x_0, x_1) = (x_0, g_1(x_1))$. Applying the existing rules in Figure 1 to such g we obtain $f(x_0, x_1) = h(x_0, g_1(x_1))$, as desired.

B. Partial Specifications

It may appear at first that the techniques presented here only work when we are given a complete specification of a problem as a function from inputs to outputs. We next show that the framework also supports enforcing arbitrary partial specifications (properties). Indeed, suppose we have a desired specification relation $r \subseteq A \times B$. We view it as a function $f' : A' \rightarrow C'$ where A' is $A \times B$ and C' is $\{0, 1\}$. We then define g to make appropriate transformations on the elements of A, and, for example, pass the elements of B unchanged. Then synthesis of h finds the combination of the outputs of g that enforces the desired properties f', which is again special case of synthesis in our framework.

C. Output Components and Synthesis in Arbitrary Context

So far we considered a problem where given components (g) pre-process the input, which then feeds into the function h that we need to synthesize. It is natural to consider a dual question (see Figure 2): we are given components k that will post-process the result, and we need to synthesize inputs for such components. This problem turns out to be directly

$$\frac{\begin{array}{c} f : A \to C \qquad h : A \to B \qquad k : B \to C \\ \forall x, y.\ I(x, y) \leftrightarrow k(y) = f(x) \\ \forall a.\ (\exists y. I(a, y)) \to I(a, h(a)) \end{array}}{\forall x. f(x) = k(h(x))}$$

$$\frac{\begin{array}{c} \text{vars}(F) \subseteq \{x\} \qquad \text{vars}(K) \subseteq \{y\} \\ [\![x\ \ \langle K = F \rangle\ \ y]\!] \vdash \langle P \mid H \rangle \end{array}}{[\![x\ \ \langle z = F \rangle\ \ z]\!] \vdash \langle P \mid K[y := H] \rangle}\ \text{O-COMP-S}$$

Fig. 2. When components apply before output, we need no interpolation

expressible using synthesis procedures framework, without a quantified synthesis condition. Figure 2 summarizes this case using the semantic rule and the corresponding syntactic synthesis procedure counterpart. In general, the components directly feed into the synthesis procedure invocation. Having pre- and post- processing components simultaneously is therefore solved using the same technique as in the case of pre-processing components alone.

D. Synthesis in Arbitrary Context and Repair

We have concluded that we can do synthesis of missing components that are fed arbitrary inputs, and whose outputs are processed in an arbitrary way. We can therefore solve for h constraints of the form $\forall x.k(h(g(x))) = f(x)$, where f can either check the property or compute value of any other desired type. Such generality enables us to use our framework to repair a given function in two steps: identify the error component, replace it with the unknown component h, then solve for h to enforce the desired constraints. This formulation may help generalize techniques used to solve the engineering change order (ECO) problem [15] to unbounded domains.

E. Synthesizing Multiple Components

Both the argument and the result of h can be a tuple. Therefore, we are able to solve synthesis problems of the form

$$\forall x.k(h_1(g'_1(x), ..., g'_n(x)), ..., h_m(g'_1(x), ..., g'_n(x))) = f(x)$$

This means that we can solve for any number of unknown components. However, note that the results of all components of g are fed into each unknown component. It may be desirable to restrict the inputs of h_i to only a subset of the variables x,

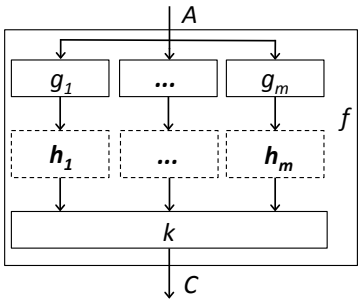


Fig. 3. Our method also handles the more general case

solving instead the problem of the form, (for some different component functions g_j):

$$\forall x.k(h_1(g_1(x)), ..., h_m(g_m(x))) = f(x)$$

To solve such problem we interpolate the following entailment, which, as before, expresses that the result of f only depends on the intermediate results returned by all of g_j:

$$\frac{f(x_1) = z_1 \wedge \bigwedge_{j=1}^{m} g_j(x_1) = y_1^j \ \wedge\ \bigwedge_{j=1}^{m} y_1^j = y_2^j}{f(x_2) = z_2 \wedge \bigwedge_{j=1}^{m} g_j(x_2) = y_2^j \ \rightarrow\ z_1 = z_2}$$

The resulting interpolant is of the form $I(y_2^1, ..., y_2^m, z_1)$; we can easily show that it satisfies, for all x and z,

$$I(g_1(x), ..., g_m(x), z) \leftrightarrow z = f(x)$$

using an entirely analogous proof as in Section III. From such component described using a relation I we can, as before, obtain a function using a synthesis procedure. We thus obtain soundness and completeness for such synthesis of multiple components that are fed distinct parts of the input (Figure 3). In addition to the previous advantages, this generalization enables the user to encode the intuition about independence between variables into the synthesis problem.

Acknowledgements.

We thank Alan Mishchenko for pointing us to existing related work, as well as for his encouraging discussions. We also thank Philippe Suter, Barbara Jobstmann, Paolo Ienne, and Anna Petkovska for useful discussions.

REFERENCES

[1] Flener, P.: Logic Program Synthesis from Incomplete Information. Kluwer Academic Publishers (1995)
[2] Jacobs, S., Kuncak, V., Suter, P.: Reductions for synthesis procedures. In: VMCAI (2013)
[3] Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS. pp. 459–473 (2006)
[4] Jiang, J.H.R., Lee, C.C., Mishchenko, A., Huang, C.Y.R.: To SAT or not to SAT: Scalable exploration of functional dependency. IEEE Transactions on Computers 59, 457–467 (2010)
[5] Jobstmann, B., Bloem, R.: Optimizations for ltl synthesis. In: FMCAD. pp. 117–124 (2006)
[6] Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. Software Tools for Technology Transfer (STTT) (2012)
[7] Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Software synthesis procedures. CACM 55(2), 103–111 (2012)
[8] Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI. pp. 316–329 (2010)
[9] Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. In: FOSSACS. pp. 395–409 (2009)
[10] Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. CACM 14(3), 151–165 (1971)
[11] McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
[12] Smith, D.R.: KIDS: A semiautomatic program development system. TSE 16(9), 1024–1043 (1990)
[13] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
[14] Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL. pp. 313–326 (2010)
[15] Wu, B.H., Yang, C.J., Huang, C.Y., Jiang, J.H.R.: A robust functional ECO engine by SAT proof minimization and interpolation techniques. In: ICCAD. pp. 729–734 (2010)