Program Synthesis Using Dual Interpretation*

Ashish Tiwari, Adrià Gascón, and Bruno Dutertre

SRI International, Menlo Park, CA, USA.

Abstract. We present an approach to component-based program synthesis that uses two distinct interpretations for the symbols in the program. The first interpretation defines the semantics of the program. It is used to specify functional requirements. The second interpretation is used to capture nonfunctional requirements that may vary by application. We present a language for program synthesis from components that uses dual interpretation. We reduce the synthesis problem to an exists-forall problem, which is solved using the exists-forall solver of the SMT-solver Yices. We use our approach to synthesize bitvector manipulation programs, padding-based encryption schemes, and block cipher modes of operations.

1 Introduction

A program is often given a concrete semantics that forms the basis of all reasoning and analysis. This semantics is typically defined over a *concrete domain* or an *abstraction* of this concrete domain, as in type checking and abstract interpretation [4]. In first-order logic, semantics is specified by a collection of structures, but there is often a single canonical structure, such as a Herbrand model minimal in some ordering, which forms the basis of reasoning. Are there any benefits in using two or more different and incomparable structures as bases for reasoning?

Type systems in programming languages can be viewed as providing second interpretations. However, they are mostly abstractions of the concrete semantics. Examples of second interpretations unrelated to the concrete semantics can be found in language-based security where ideas such as security-type systems and, more generally, semantic-based security are explored [14]. Many security properties, for example noninterference, are not concerned with the functionality of a program, but *how* it implements such functionality in the presence of a malicious adversary. The analysis of such *nonfunctional properties* usually benefits from having alternate semantics modeling the attacker's view of the program.

We use dual interpretations for performing program synthesis. We illustrate our approach on synthesis of cryptographic schemes. A correct cryptographic scheme must satisfy two different properties. First, every encryption scheme

* This work was sponsored, in part, by ONR under subaward 60106452-107484-C under prime grant N00014-12-1-0914, and the National Science Foundation under grant CCF-1423296. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the funding agencies.

should have a corresponding decryption scheme. This functional correctness property can be decided using the concrete semantics of the program. Second, we must guarantee that the encryption scheme is secure (in some attacker model). This property is not functional and it is difficult to specify using the concrete semantics. Instead, it is sometimes possible to reason (conservatively) about security properties using a second, completely different, meaning of the program. This observation motivates the dual interpretation approach of this paper.

Ostensibly, the prospect of having two different semantics for programs seems to be a potentially troublesome idea. However, in theory, it is not much different from having just one concrete semantics since one could merge the two semantics by considering the product of the two domains. For reasoning though, it is still beneficial to consider the two semantics separately.

Our goal is to automatically generate correct programs using components or functions from a library. The synthesized program must satisfy both functional and nonfunctional requirements. We use a primary concrete semantics to specify the functional requirements and an alternate semantics to specify the nonfunctional requirements. The second interpretation is also used to restrict the set of candidate programs. We present a language for writing program sketches and specifying both types of requirements. We solve the synthesis problem by compiling it to an exists-forall formula, which our tool currently solves using the exists-forall solver of Yices [5]. We provide experimental evidence of the value of the language and our synthesis approach by presenting a collection of examples that were automatically synthesized using our tool.

1.1 Related Work

Our work is inspired by recent progress in the area of program synthesis. Synthesizing a program from an abstract specification is not achievable in practice but *template-based* synthesis has shown a lot of promise [1,16,17]. In this approach, the designer provides a *template* that captures the shape of the intended solution(s) together with the specification. A synthesis algorithm fills in the details. This general idea has been successfully applied to several domains. For example, imperative programs can be obtained from a given sketch, as long as their intended behavior is also provided [15]; efficient bitvector manipulations can be synthesized from naïve implementations [10]; agent behavior in distributed algorithms can be synthesized from a description of a global goal [8]; and deobfuscated code can be obtained using similar ideas [11]. Although all these applications rely on template-based synthesis, different synthesis algorithms are used in different domains. Logically, most of the synthesis algorithms are solving an *exists-forall* problem.

Recently, we have implemented an exists-forall solver as part of the SMT-solver Yices [5,7]. In this paper, we present a language for specifying sketches, which are partially specified programs, but unlike any previous work on synthesis, we use two different interpretations for the program symbols. We perform synthesis by explicitly generating an exists-forall formula in Yices syntax and using Yices to solve it.

Component-based program synthesis problem was formulated in [10], but the interest in [10] was only on functional requirements. Here, we also consider non-functional requirements, which forces us to reason with two different semantics of the same program. We use some of the benchmarks from [10] in this paper.

2 Component-Based Program Synthesis

We assume that programs are constructed from a library of components. We are interested in constructing straight-line programs using the library. A straight-line program can be viewed as a term over the signature of the library.

Let Σ be a signature consisting of constant and function symbols. Let $\texttt{Vars}$ denote a set of (input) variables. Let $\texttt{Terms}(\Sigma, \texttt{Vars})$ denote the set of all terms defined over the signature Σ and variables $\texttt{Vars}$. A term t in $\texttt{Terms}(\Sigma, \texttt{Vars})$ naturally corresponds to a straight-line program whose inputs are the variables occurring in t. For example, the term $f(g(x), x)$ corresponds to the following program:

$$\texttt{input } x;\ \ y_1 := g(x);\ \ y_2 := f(y_1, x);\ \ \texttt{output } y_2$$

We give meaning to programs using a *structure* $(\texttt{Dom}, \texttt{Int})$ where the domain $\texttt{Dom}$ is a nonempty set, and the interpretation $\texttt{Int}$ maps every constant $c \in \Sigma$ to an element $c^{\texttt{Int}} \in \texttt{Dom}$ and every function symbol $f \in \Sigma$ with arity, say, 2, to a concrete function $f^{\texttt{Int}} : \texttt{Dom} \times \texttt{Dom} \mapsto \texttt{Dom}$.

In the program-synthesis terminology, the symbols Σ and their interpretation I form the *library* of *components*.

We can extend the interpretation $\texttt{Int}$ to $\texttt{Int}'$ by adding interpretation for (input) variables $\texttt{Vars}$. Now, the interpretation $\texttt{Int}'$ is extended to terms t over $\texttt{Terms}(\Sigma, \texttt{Vars})$ in the natural way, and we denote the interpretation of t by $t^{\texttt{Int}'}$.

Example 1. Let $\Sigma = \{(f : 2), (g : 2), (h : 1), (c : 3), (d : 2), (e : 0)\}$ be a signature. Let $\texttt{Dom}$ be the set of bitvectors of an arbitrary but constant length k, and let $\texttt{Int}$ be the function

$$\begin{array}{lll} \texttt{Int}(f) = \texttt{bv-xor} & \texttt{Int}(g) = \texttt{bv-and} & \texttt{Int}(h) = \texttt{bv-neg} \\ \texttt{Int}(c) = \texttt{ite} & \texttt{Int}(d) = \texttt{bv-sgt} & \texttt{Int}(e) = 0 \ldots 01 \end{array}$$

where $\texttt{bv-xor}$ is bitwise $\texttt{xor}$, $\texttt{bv-and}$ is bitwise $\texttt{and}$, $\texttt{bv-neg}$ is the negative in 2s complement, $\texttt{ite}$ is $\texttt{if-then-else}$, $\texttt{bv-sgt}$ is $\texttt{signed greater-than}$, and $0 \ldots 01$ is the bitvector representing 1. Consider the term $s = c(d(x, y), x, y)$. Under the meaning defined by the structure $(\texttt{Dom}, \texttt{Int})$, s corresponds to a program computing the maximum of two binary integers of length k. Note that the term s is *equivalent* to the term $t = f(g(f(x, y), h(d(x, y))), y)$ under the semantics defined by $(\texttt{Dom}, \texttt{Int})$, but t does not use c.

2.1 Functional Requirements

One of the requirements for synthesis is that the synthesized program have the same interpretation as a given (desired) program.

Let $f_{\mathtt{spec}} : \mathtt{Dom}^n \mapsto \mathtt{Dom}$ be a concrete function over the domain $\mathtt{Dom}$. Let $\mathtt{Vars} = \{i_1, \ldots, i_n\}$ be a set of n variables. The first requirement is that we should synthesize a term $t \in \mathtt{Terms}(\Sigma, \mathtt{Vars})$ that satisfies the condition that, for all $e_1, \ldots, e_n \in \mathtt{Dom}$, it should be the case that

$$t^{\mathtt{Int}'} = f_{\mathtt{spec}}(e_1, \ldots, e_n) \quad \text{where } \mathtt{Int}' := \mathtt{Int} \circ \{i_1 \mapsto e_1, \ldots, i_n \mapsto e_n\} \quad (1)$$

The reader may wonder about the form of the description of $f_{\mathtt{spec}}$, and ask why we do not just use the description of $f_{\mathtt{spec}}$ to construct t. The reason is that $f_{\mathtt{spec}}$ may be described with symbols that are not available for constructing t. For example, $f_{\mathtt{spec}}$ may be specified by a program that contains "`if-then-else`" constructs, whereas the library `Int` for synthesizing t may not contain such a function (as in Example 1). In general, we can assume that only a subset of the functions in the library are available for synthesis, whereas all functions in the library can be used for specifying $f_{\mathtt{spec}}$. Furthermore, t will be required to be of a given size, which will often be less than the size of $f_{\mathtt{spec}}$.

2.2 Nonfunctional Requirements

To capture nonfunctional requirements, we assume that we have another structure $(\mathtt{Typ}, \mathtt{TCC})$, where `Typ` is a domain and `TCC` is an interpretation where

- `TCC` maps a constant $c \in \Sigma$ to a subset of `Typ`, and
- `TCC` maps a function $f \in \Sigma$ of arity i to a subset of $\mathtt{Typ}^{i+1}$.

Variables, just like constants, are interpreted as subsets of `Typ`. We can extend the interpretation `TCC` by including the interpretation for (input) variables `Vars` to get $\mathtt{TCC}'$. We can now extend the interpretation $\mathtt{TCC}'$ to all terms of $\mathtt{Terms}(\Sigma, \mathtt{Vars})$ as follows:

- if $t_i^{\mathtt{TCC}'} \subseteq \mathtt{Typ}, i = 1, \ldots, n$ are the interpretations of terms $t_1, \ldots, t_n$, then $f(t_1, \ldots, t_n)^{\mathtt{TCC}'}$ is the set

$$\{y \in \mathtt{Typ} \mid (y_1, \ldots, y_n, y) \in f^{\mathtt{TCC}}, y_i \in t_i^{\mathtt{TCC}'} \text{ for } i = 1, \ldots, n\}$$

Without loss of generality, we assume that we are synthesizing a function with n inputs, say $i_1, \ldots, i_n$, and one output (that is, $f_{\mathtt{spec}}$ has arity n). The nonfunctional requirement ϕ is a subset of $\mathtt{Typ}^{n+1}$. Formally, a synthesized program $t(i_1, \ldots, i_n)$ satisfies the nonfunctional requirement ϕ if *there exist* $(e_1, \ldots, e_n, e) \in \phi$ such that

$$e \in t^{\mathtt{TCC}'} \text{ for the interpretation } \mathtt{TCC}' := \mathtt{TCC} \circ \{i_1 \mapsto \{e_1\}, \ldots, i_n \mapsto \{e_n\}\} \quad (2)$$

Here $\mathtt{TCC}'$ is the same as interpretation `TCC` except for the interpretations of $i_1, \ldots, i_n$.

Remark 1. *Type* information in programming languages can be captured using the second interpretation structure ($\texttt{Typ}, \texttt{TCC}$). In such a case, the predicates in $\texttt{TCC}$ will be *type correctness conditions*. But the second interpretation structure is more general, and it need not be an abstraction of the concrete domain. The second interpretation serves two purposes. First, it can be used to encode nonfunctional requirements. Second, it can be used to prune the synthesis search space, since a program (term) that can not be "typed" can be pruned early.

Example 2. Consider the signature Σ from Example 1. We can define a second interpretation ($\texttt{Typ}, \texttt{TCC}$), where $\texttt{Typ} := \{\texttt{true}, \texttt{false}\}$ and for all symbols F in Σ of arity n, let $\texttt{TCC}(F) = \{(b_1, \ldots, b_n, b) \in \texttt{Typ}^{n+1} \mid b = \bigvee_i b_i\}$. If the input variables x, y are interpreted as $\{\texttt{true}\}$, then a term t will be interpreted as $\{\texttt{true}\}$ iff it contains a variable. A ground term, such as $f(e, e)$, will get an interpretation $\{\texttt{false}\}$. This can help us identify (and prune out) programs that do not use the input(s).

2.3 Problem Definition

We now define the component-based program synthesis problem with functional and nonfunctional requirements as follows.

Definition 1 (Program Synthesis with Dual Requirements). *Given two structures* ($\texttt{Dom}, \texttt{Int}$) *and* ($\texttt{Typ}, \texttt{TCC}$) *that provide two different interpretations for the symbols in Σ, a size requirement N, a functional requirement $f_{\texttt{spec}} : \texttt{Dom}^n \mapsto \texttt{Dom}$ and a nonfunctional requirement $\phi \subseteq \texttt{Typ}^{n+1}$, the component-based program synthesis problem seeks to find a term $t \in \texttt{Terms}(\Sigma, \{i_1, \ldots, i_n\})$ of size N such that for all $e_1, \ldots, e_n \in \texttt{Dom}$ the condition in Equation 1 holds, and for some $(e_1, \ldots, e_n, e) \in \phi$, the condition in Equation 2 holds.*

3 Synthesis Approach

The program-synthesis problem formulated in Definition 1 can be reduced to an exists-forall formula, which is then solved using an off-the-shelf solver.

Let $\texttt{subterms}(t)$ denote the set of all subterms of the term t. Henceforth, fix $\texttt{Vars} = \{i_1, \ldots, i_n\}$.

Consider the program synthesis with dual requirements problem in Definition 1. The problem can be rewritten in logical notation as follows:

$$\begin{aligned}
&\exists t \in \texttt{Terms}(\Sigma, \texttt{Vars}) : \texttt{size}(t) = N \;\wedge \\
&(\exists \tau : \texttt{subterms}(t) \mapsto \texttt{Typ} : \\
&\;(\forall s \in \texttt{subterms}(t) : s = f(s_1, \ldots, s_m) \Rightarrow (\tau(s_1), \ldots, \tau(s_m), \tau(s)) \in f^{\texttt{TCC}}) \;\wedge \\
&\;(\tau(i_1), \ldots, \tau(i_n), \tau(t)) \in \phi) \;\wedge \\
&(\forall e_1, \ldots, e_n \in \texttt{Dom} : f_{\texttt{spec}}(e_1, \ldots, e_n) = t^{\texttt{Int}'}) \qquad (3)
\end{aligned}$$

where $\texttt{Int}' := \texttt{Int} \circ \{i_1 \mapsto e_1, \ldots, i_n \mapsto e_n\}$. Clearly, a witness for t in this formula is a solution to the synthesis problem.

```
(isolateRightmostOne_sketch
 (comment "Isolate rightmost 1 bit in the input bitvector")
 (decls
  (define-type word (bitvector 5))
  (define fbvand::(-> word word word) (lambda (x::word y::word) (bv-and x y)))
  (define fbvneg::(-> word word) (lambda (x::word) (bv-neg x)))
  (define frightmost1::(-> word word) (lambda (x::word)
   (ite (bit x 0) (mk-bv 5 1) (ite (bit x 1) (mk-bv 5 2) (ite (bit x 2) (mk-bv 5  4)
   (ite (bit x 3) (mk-bv 5 8) (ite (bit x 4) (mk-bv 5 16) (mk-bv 5 0))))))))
  (define-type typ bool)
  (define tbvand::(-> typ typ typ bool) (lambda (x::typ y::typ z::typ) (= z (or x y))))
  (define tbvneg::(-> typ typ bool) (lambda (x::typ y::typ) (= y x))))
 (parameters na)
 (library (bvand 2) (bvneg 1))
 (blocks
  (Lx 1 ((input x::true)))
  (l1 na ((bvand (Lx -) (Lx -)) (bvneg (Lx -))))
  (spec 1 ((rightmost1 (Lx) ))))
 (ensure (and (= (value l1 na) (value spec 1)) (= (type l1 na) true)))
```

Fig. 1. A small Synudic example that can be used to synthesize a two-line program for isolating the rightmost 1 in a bitvector.

We define the size $\texttt{size}(t)$ of a term t to be the cardinality of $\texttt{subterms}(t)$; that is, the size of a minimal DAG representing t. Since we assume that Σ is finite, there are only finitely many terms of size N, and hence the first existential corresponds to a finite search. Since the cardinality of $\texttt{subterms}(t)$ is N, the second existential reduces to existence of N elements of Typ. The next $\forall$ quantifier is over a finite set and hence it is just a short-hand for a large conjunction. Finally, the last $\forall$ quantifier is over n elements of Dom, and thus, we map our synthesis problem to an exists-forall problem in the theory of Dom and Typ.

To increase expressiveness and improve scalability, we need an approach that allows a user to prune the search space for t as much as possible. We have designed a language that not only allows users to specify the program synthesis problem with dual requirements (Definition 1), but also allows users to constrain the search space. We briefly describe this language next.

3.1 Synudic: A Language for Synthesis Using Dual Interpretations on Components

Synudic (Synthesis using dual interpretation on components) is a language for specifying program synthesis problems with dual requirements (Definition 1). It also allows users to provide additional restriction on the structure of the program to be synthesized.

We call a well-formed Synudic term a *sketch* since it is not really an executable program, but an incomplete program with a specification. For ease of parsing, a Synudic sketch is an S-expression. Rather than provide details on the language, we illustrate it using an example here.

Given a bitvector x, consider the function frightmost1 that returns a bitvector that has 1 only at the position of the rightmost 1 in x. For example, $\texttt{frightmost1}(10110) = 00010$. Figure 1 shows a small Synudic sketch that can be used to synthesize a two-line program for computing `frightmost1`. It contains the following information:

Σ: The library Σ, defined on Line 14, consists of a binary symbol `bvand` and a unary symbol `bvneg`.

`Dom`: The domain `Dom`, defined on Line 4, consists of bitvectors of length 5.

`Int`: The interpretation `Int`, defined on Lines 5-6, consists of two Yices functions, `fbvand` and `fbvneg` that provide meaning to the two symbols in Σ. Essentially, `fbvand` computes a bitwise "and" and `fbvneg` computes the negative (in 2s complement notation).

`Typ`: The second domain `Typ`, defined on Line 10, consists of the Booleans.

`TCC`: The second interpretation `TCC`, defined on Lines 11-12, consists of two Yices functions, `tbvand` and `tbvneg` that provide (second) meaning to the two symbols in Σ.

Sketch: The program sketch, defined on Lines 15-18, consists of three blocks.
Line 16: The first block, labeled `Lx`, has 1 line that outputs the value of the input variable x. We also have $\texttt{TCC}(x) = \{\texttt{true}\}$.
Line 17: The second block, labeled `l1`, has na lines, where na is a parameter (that we will set to 2 since we are interested in synthesizing a two line program) and each line can use either the `bvand` function or the `bvneg` function. The arguments of the two functions can come from block `Lx` or from previous lines of this block, which is denoted by the list "`(Lx -)`".
Line 18: The third block, labeled `spec`, has 1 line that computes the value `frightmost1` on the input x.

Requirements: The requirement, defined on Line 19, says that the value computed on Line 1 of block `spec` is equal to the value computed on line na of block `l1` (functional correctness). Moreover, the type computed on line na of block `l1` is equal to `true` (nonfunctional requirement).

The specification function, `rightmost1`, is defined on Lines 7-9 using nested "if-then-else" calls. The Boolean "type" attached to each value just denotes whether the input was syntactically used to compute that value.

Remark 2. Our language is designed so that it can be used to verify a concrete program, as well as, synthesize a correct program from a library of pre-defined functions. A concrete straight-line program can be written using blocks of length 1 in which there is just one option for the right-hand side expression. On the other extreme, an arbitrary straight-line program of length n over a library containing functions $f_1, \ldots, f_m$ can be written as

$$(L1\ \ n\ \ ((f_1\ \ (L0\ -)\ \ (L0\ -))\ \ (f_2\ \ (L0\ -)\ \ (L0\ -)) \ldots (f_m\ \ (L0\ -)\ \ (L0\ -))))$$

where $L0$ is the block generating the inputs. When performing synthesis, finding one program from the set of *all* n line programs can be difficult. Our language allows users to specify program search space that falls any where in between these two extremes.

3.2 From Synudic Sketches to Yices ∃∀ Formulas

Given a Synudic sketch, we have a tool that generates the corresponding exists-forall formula (shown in Equation 3) in Yices syntax. Note that Synudic defines `Dom` and `Typ` as types in Yices, and gives interpretations as Yices functions. Moreover, the program sketch in Synudic also fixes the size of the term t to be synthesized. The additional constraints imposed by the block structure are also added to the exists-forall formula—in fact, all these additional constraints are on the existential variables. We skip the details of the translation into a Yices formula because it is straightforward. In fact, the translation borrows several ideas from the translation proposed in [10] and extends them to handle the dual interpretations and block structure restrictions, which were both absent in [10].

Our tool calls the exists-forall solver of Yices on the generated ∃∀ formula. If there is a solution, the tool outputs the model for the existential variables, which can be used to obtain the concrete program. By giving an appropriate command-line argument, the tool can also search for alternate (more than one) solutions for the same sketch.

We next describe case studies from two domains - synthesis of bitvector manipulation tricks and synthesis of cryptographic schemes.

4 Bitvector Manipulation Programs

As a baseline, we evaluate our approach on bitvector manipulation benchmarks from [10,18]. The goal of these experiments is to show that (a) synthesis benchmarks that have been used before can be specified in the Synudic language, and (b) features supported by Synudic can be used to speed-up the synthesis process.

A simplified version of one our benchmark examples was presented in Figure 1. (The version used in our experiments had a larger library.) We note a few salient features of all the bitvector synthesis benchmarks.
(1) First, we use bitvectors of length 5 as `Dom`. It turns out that the algorithms that are synthesized to work on bitvectors of length 5 also work on bitvectors of arbitrary length. This observation was already made in [10]. We just note here that our language allows the user to set `Dom` to any type (supported by Yices).
(2) We use the usual bitvector operations, such as bitwise or, and, xor, as well as arithmetic functions on bitvectors, such as add and subtract, in the library. Certain examples also need functions that perform bitvector comparison, shift right, and division. We included them in the library whenever they were needed.
(3) Subtracting 1 is a common operation. We have two options: either we can include a subtract 1 operation as a library primitive, or we can include the subtraction operation and a function that generates the constant 1 in the library. Our language can support both choices. Using the former option usually speeds up the synthesis process.
(4) We used the Booleans as `Typ`. The Boolean value associated to a program variable keeps track of whether "the input was used to compute the value of that program variable", as shown in Example 2. For the bitvector examples, the

name	function $x(,y) \mapsto z$	#lines	#lib	time	#lib	time	timet
rightmost 1 off	$u10^* \mapsto u00^*$	3	6	0.24	8	0.5	0.5
isolate rightmost 1	$u10^* \mapsto 0^*10^*$	2	7	0.18	9	0.2	0.2
average	$z = \frac{x+y}{2}$	4	4	2.9	7	27	5.4
mask for 10*$	$u10^* \mapsto 011^*$	3	7	0.2	9	0.2	0.5
Maximum	$z = \max(x, y)$	4	4	77	7	238	86
turnoff $1^+0^*$$	$u1^+0^* \mapsto u0^+0^*$	5	6	21	8	102	2
next# same#1s	$\min z$ s.t. $z > x,\ z\|_1 = x\|_1$	8	5	154	6	$\frac{500}{TO}$	54

Table 1. Bitvector benchmarks: Column `#lines` is the number of lines in the synthesized program, `#lib` is the number of functions in the library used for synthesis, `time` denotes the time (in seconds) taken for the tool to synthesize the program, and `timet` denotes the time taken when using a second interpretation to prune search space.

second interpretation was not strictly required (since there was no nonfunctional requirement).

We present the results from bitvector benchmarks in Table 1. Synthesizing longer programs takes longer, and increasing the library size usually increases the time taken for synthesis (Columns 5 and 7), but in some cases, the rise is steep (third example computing "average"). To evaluate the benefit of pruning using the second interpretation, we added a second interpretation to enforce that certain library components are used (at most) once, and the running times with the second interpretation added are shown in the last column in Table 1. As expected, our running times in Column 7 are comparable to those reported in [10]. In some cases, our tool synthesized "new" procedures that were semantically equivalent variants of the known procedures, see [9] for such examples.

5 Cryptographic Constructions

We now provide examples of how dual interpretations are useful for the synthesis of cryptographic constructions. We first provide an example from public key cryptography inspired by the work in [2] that consist on synthesizing padding schemes. Our second example is related to symmetric key encryption, and builds upon the work presented in [13].

5.1 Synthesis of Padding-based Encryption Schemes

In public key cryptography, padding is the process of preparing a message for encryption. A modern form of padding is OAEP, which is often paired with RSA public key encryption. Padding schemes, and in particular OAEP, satisfy the goals of (1) converting a deterministic encryption scheme, e.g. RSA, into a probabilistic one, and (2) ensuring that a portion of the encrypted message cannot be decrypted without being able to invert the full encryption.

```
(oaep_sketch
 (decls ...)
 (parameters na nb)
 (library (G 1) (H 1) (oplusr 2) (oplus 2) (identity 1))
 (blocks
  (lm 1 ((input m::(bool-to-bv false false false false true))))
  (lr 1 ((input r::(bool-to-bv false false false true false))))
  (l1 na ( (oplusr (lm lr) (-)) (G (lr -)) (H (lm -))))
  (l2 2  ( (identity (l1 lr) ) ))
  (l3 nb ( (oplus (l2 -) (l2 -)) (H (l2 -)) (G (l2 -)) )))
 (ensure (and (= (value lm 1) (value l3 nb))
              (isrand (type l2 1)) (isrand (type l2 2)))))
```

Fig. 2. Sketch used for synthesizing various padding-based encryption schemes. The full example can be found at [9].

Inspired by the success of the tool Zoocrypt in synthesizing padding-based encryption schemes [2] (and their corresponding security proofs), we used our synthesis tool for exploring the same space.

Figure 2 shows part of the sketch that we used. The full sketch of this example is available at [9]. The library of components defined by Σ and `Int` in this example consists of two unary hash functions, G and H, a binary *xor* function (called `oplus` in Figure 2), a slight variant of *xor* called `oplusr`, and the identity function. Padding with 0 is not modeled explicitly. It is added as a post-processing step to make the hash functions applicable on its arguments.

The sketch in Figure 2 has two inputs—the message m in block lm line 1, and a random number r in block lr line 1. This is followed by a straight-line code block $l1$ of length na that constructs the padding scheme. It is allowed to use the hash functions and the *xor* function. Two of the values computed in block $l1$ (including the random number r) are picked in block $l2$ to be concatenated, encrypted and sent on the network. The block $l3$ decodes the messages received from block $l2$. The decoding block is of length nb and it can use the hash functions and the *xor* function.

As expected, we encoded the desired security properties of a padding-based encryption scheme using nonfunctional requirements. As `Typ` we used bitvectors of length 5, since that was enough to encode our type constraints:
(a) The first bit keeps information about the size of the computed value. This information is necessary to produce type correct programs, since we have hash functions mapping bitvectors of one size to another.
(b) The second bit is set if the data value is essentially the same as a random value in its domain. It is difficult to carry forward this information precisely, so we use conservative typing rules to update the value of the second type-bit during each operation.
(c) The third and fourth bits are set if the top function application is the hash

$$f(G(r)||(G(r) \oplus m))$$
$$f(r||(G(r) \oplus m))$$
$$f(G(r \oplus H(m))||(G(r \oplus H(m)) \oplus m))$$
$$f((r \oplus H(m))||(G(r \oplus H(m)) \oplus m))$$
$$f((G(r) \oplus m)||(H(G(r) \oplus m) \oplus r))$$

Fig. 3. Some automatically synthesized padding-based encryption schemes.

function G and H, respectively. This information is used to update the second bit of the type.
(d) The fifth bit is set if the top function application is the *xor* function. This information is used for the same purpose as the previous two type-bits.

Finally, in the ensure section we state (a) the functional requirement: the result of decoding (written as `(value l3 nb)`, the value on line `nb` in block `l3`) should be equal to the message `m` (written as `(value lm 1)`, the value on line 1 in block `lm`), and the (b) nonfunctional requirement: the two values that are transmitted, namely the value on lines 1 and 2 of block `l2`, should essentially be random; that is, the second bit of their respective type values from `Typ` should be set. The type value of line 1 in block `l2` is written as `(type l2 1)`.

The declarations part of the sketch in Figure 2 can be found at [9]. We note two things. First, we used fixed length bitvectors as `Dom`. The length choice is arbitrary: larger bitlengths would mean more computational resources would be required to solve the synthesis problem, but smaller bitlengths could lead to synthesis of schemes that do not work for arbitrary sizes. Second, the interpretations of H and G had to be concretized to bitvector functions, but they had to be picked carefully so that they satisfy (exactly) the algebraic relations the actual functions satisfy. This may not be possible always, in which case one should choose interpretations that are likely to lead to correct solutions. In such cases, a post-processing security verification tool will be needed to verify the synthesized schemes.

We used our tool to synthesize different padding schemes using different values for the two parameters `na` and `nb`. We can use the tool to generate different solutions for the same values of the parameters.

Some example synthesized schemes are shown in Figure 3. Again, we do not show the padding with 0 that is required to make arguments reach the required bitvector length. Note that the OAEP scheme [3] was also generated (using $\texttt{na} = 4$ and $\texttt{nb} = 4$—it is the last schemes in Figure 3. But smaller padding-based schemes were also found by the tool. Similar schemes have also been reported in [2].

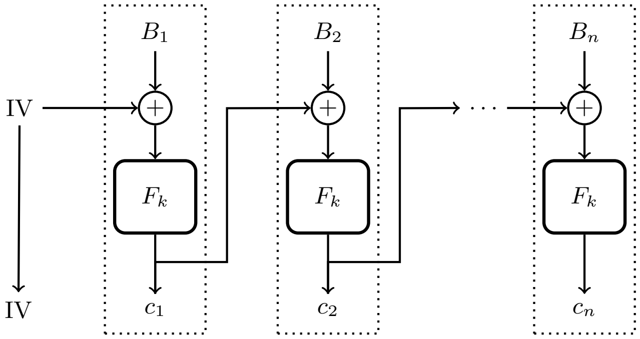

Fig. 4. The CBC mode of operation for the encryption of an n-block message. The dotted boxes correspond to the multiple copies of the block processing procedure.

5.2 Synthesis of Block Ciphers Modes of Operation

A block cipher consists of one algorithm for encryption and one for decryption implementing functions $F : \{0,1\}^l \times \{0,1\}^{l_k} \to \{0,1\}^l$ and $F^- : \{0,1\}^l \times \{0,1\}^{l_k} \to \{0,1\}^l$, respectively. F and F^- satisfy that (i) given a *block* $B \in \{0,1\}^l$ and a key $k \in \{0,1\}^l_k$, $F(B,k)$ and $F^-(B,k)$ return a permutations $F_k(B)$ and $F^-_k(B)$ of B, and (ii) for every $k \in \{0,1\}^l_k$ and block $B \in \{0,1\}^l$, $F^-_k(F_k(B)) = B$. An example of block cipher is the standardized AES, for which $l = 128$.

Roughly speaking (see [12] for a formal definition), a block cipher (F, F^-) is secure against the so-called chosen plaintext attacks (in the standard model) if, fixed a random key k, an attacker allowed to query F_k has negligible probability of distinguishing F_k from a randon permutation, given certain limitations on the computational power of the attacker and the number of times F_k can be queried.

A *mode of operation* is a pair of algorithms that features the use of a symmetric block cipher algorithm (F, F^-), e.g. AES, to encrypt/decrypt amounts of data larger than a block. A secure mode of operation must provide the same level of security than its associated block cipher. For example, the encryption algorithm of the popular Cipher Block Chaining (CBC) mode is depicted in Figure 4. CBC, when equipped with a secure block cipher, provides IND$-CPA security, i.e. an attacker cannot distinguish its output from an uniformly random string with significant probability (under certain constraints on the computational power of the attacker). Note that CBC encryption consists of an initialization algorithm, where a random initialization vector IV is produced, followed by n copies of a block processing algorithm, while exactly one value is fed from one copy to the next one. This structure is common to many of the popular modes of operation.

Most of the previous approaches to the formal verification and synthesis of block cipher modes of operation (and certainly the ones considered in this paper) build upon the observation that these kind of programs can be constructed using a limited set of operations such as xor, concatenation, generation of random values, and evaluation of the block cipher.

Recent effort in the automation of the analysis of block cipher modes include [6,13]. In contrast to [6], which suffers from the limitation that the analyzed mode must operate on a *fixed* number of blocks, the work in [13] models the operation that is carried out when encrypting a *single* block, exploiting the common structure of block cipher modes of operation mentioned above. In [13], the encryption algorithm of a mode of operation is described as a pair of straight-line programs (`Init`, `Block`). `Init` models the initialization phase of the mode of operation. In the case of the CBC mode of Figure 4, `Init` would correspond to the generation of the random value IV. On the other hand, `Block` corresponds to the algorithm that, given a value coming from the previous iteration (or the initialization phase) and a certain message block m, produces the ciphertext for m and the value to be fed to the next iteration of the mode of operation. For the CBC mode, the different instances of `Block` correspond to the subalgorithms in dotted boxes in Figure 4. While `Init` is very simple in that, roughly speaking, it may contain only a random number generation operation, `Block` might contain an arbitrary number of xor operations and evaluations of F_k for a fixed value of k. A further relevant structural restriction in the straight-line programs `Init` and `Block` is that the output of every operation in the program must be used exactly once in the rest of the program, with the exception of an additional operation called dup implementing the identity function and whose output must be used twice.

As main contribution in [13], the authors present a type system T that guarantees that type correct modes (`Init`, `Block`) encode secure modes of operation. Then, synthesis of secure modes is performed by enumerating straight-line programs satisfying the constraints above and filtering out the ones that are not type correct w.r.t. T. This check is implemented by means of an SMT solver. An ad hoc procedure is used to further guarantee that the resulting mode of operation admits a decryption algorithm.

In the example presented in this section, we encoded the synthesis approach from [13] as a program synthesis with dual requirements problem (Definition 1) Instead of separately filtering modes of operation that are not decryptable as done in [13], we encoded the existence of a decoding algorithm as a *functional requirement.* That has the advantage that encryption algorithms are synthesized together with their corresponding decryption procedure. Moreover, the constraint that `Init` and `Block` must be type correct w.r.t. T can be naturally encoded as a *nonfunctional requirement* in our language.

While reducing the synthesis of block ciphers modes of operation to the program synthesis with dual requirements problem has many advantages, our approach also suffers from some limitations with respect to the one in [13]. The main limitation of our encoding is that, whereas the approach of [13] is completely symbolic, we needed to provide a specific domain `Dom` and interpretation `Int` for every operation (including the permutation F_k). For `Dom` we chose bitvectors of length 5. While xor, dup1, and dup2 have natural operations in

Parameters			Modes	Time(s)
na	nb	nc		
2	4	3	CBC	3.25
2	6	3	CBC* OFB*	6.07
2	6	4	CBC*	22.9
2	6	5	CBC OFB* CFB	5.77

Parameters			Modes	Time(s)
na	nb	nc		
2	7	6	OFB variant CBC variant	6.34
2	8	5	CBC* OFB* CFB*	39.47
2	9	5	PCBC OFB variant	109.82

Fig. 5. Results of the synthesis of block cipher modes of operation using Synudic.

the domain of bitvectors. The interpretation of F should be picked carefully so that it satisfies (exactly) the algebraic relations the actual functions satisfy. We picked left-rotation by two for the interpretation of F. Although in principle a bad election for such representation might cause invalid schemes to be accepted as decryptable, this can be easily avoided in many cases. The full sketch used for this example can be found at [9].

Using our tool we could synthesize the well-known modes ECB, OFB, CFB, CBC, and PCBC, also automatically found in [13], as well as some variants of those. The tables of Figure 5 show the size parameters needed to obtain each of them. The reported times corresponds to a complete exploration of the search space that correspond to the parameters. For example, the second row of the first table means that, with parameters $na = 2$, $nb = 6$, $nc = 3$, it took our tool 6.07 seconds to conclude that *exactly* two instances of the sketch are secure and decryptable modes of operation. The modes marked with an asterisk (*) correspond to redundant variants of the corresponding mode.

6 Conclusion

We presented an approach for program synthesis that relies on using two different interpretations for the program variables. The dual interpretation approach enables specification of both functional and nonfunctional requirements. It also helps in pruning the synthesis search space. We applied our approach to synthesize nonintuitive bitvector manipulation tricks and secure cryptographic protocols.

We first defined a sketching language that can be used to specify library functions from which a scheme needs to be generated. The language features can be used to prune the search space of all valid programs. We translate the synthesis problem in the sketching language to an $\exists\forall$ Yices formula, and use the Yices $\exists\forall$ SMT solver to solve the constraint and obtain a possible program. We used our language and the accompanying synthesis tool to the synthesis of padding-based encryption schemes and block cipher modes of operation.

The dual interpretation approach is potentially more generally useful. The second interpretation can carry information pertaining to a predicate abstraction of the program, but it can as well carry information unrelated to the concrete semantics, such as provenance or information flow.

Our current implementation is limited in several ways. First, the Yices exists-forall solver handles only bitvectors, Booleans, and linear arithmetic expressions. Hence, only these types can be used to define the two interpretations. Our synthesis language allows synthesis of only straight-line programs, and does not allow, for example, synthesis of functions that are used within other functions. Such extensions are left for future work.

References

1. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 1–17, 2013.
2. G. Barthe, J. M. Crespo, C. Kunz, B. Schmidt, B. Gregoire, Y. Lakhnech, and S. Zanella-Beguelin. Fully automated analysis of padding-based encryption in the computational model, 2013. `http://www.easycrypt.info/zoocrypt/`.
3. M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology, EUROCRYPT*, volume LNCS 950, 1994.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages, POPL 1977*, pages 238–252, 1977.
5. B. Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.
6. M. Gagné, P. Lafourcade, Y. Lakhnech, and R. Safavi-Naini. Automated verification of block cipher modes of operation, an improved method. In *Foundations and Practice of Security*, volume 6888 of *LNCS*, pages 23–31. Springer, 2011.
7. A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanovic, and S. Malik. Template-based circuit understanding. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 83–90. IEEE, 2014.
8. A. Gascón and A. Tiwari. A synthesized algorithm for interactive consistency. In *6th Intl Symp NASA Formal Methods*, number 8430 in LNCS, pages 270–284, 2014.
9. A. Gascón and A. Tiwari. Synudic: Synthesis using dual interpretation on components, 2015. `http://www.csl.sri.com/users/tiwari/softwares/auto-crypto/`.
10. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proc. ACM Conf. on Prgm. Lang. Desgn. and Impl. PLDI*, pages 62–73, 2011.
11. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proc. ICSE (1)*, pages 215–224. ACM, 2010.
12. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman and Hall/CRC Press, 2007.
13. Alex J. Malozemoff, Jonathan Katz, and Matthew D. Green. Automated analysis and synthesis of block-cipher modes of operation. In *IEEE 27th Computer Security Foundations Symposium, CSF*, pages 140–152. IEEE, 2014.
14. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
15. Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
16. Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

17. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Vijay Saraswat, and Sanjit Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.
18. Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.