Solving Exists/Forall Problems With Yices

Extended Abstract

Bruno Dutertre

Computer Science Laboratory
SRI International
Bruno.Dutertre@sri.com

Abstract

Yices now includes a solver for Exists/Forall problem. We describe the problem, a general solving algorithm, and a key model-based generalization procedure. We explain the Yices implementation of these algorithms and survey a few applications.

1 Introduction

The traditional SMT problem is to determine whether a quantifier-free formula $\Phi(x)$ is satisfiable. Some solvers can also handle first-order formulas with arbitrary quantifiers. We are concerned with a simpler case, namely, formulas of the form $\forall y.\Phi(x,y)$, where $\Phi(x,y)$ is quantifier-free. Implicitly, the variables x are existentially quantified: we are checking the satisfiability of formulas of the form $\exists x.\forall y.\Phi(x,y)$, thus, we will follow tradition and call this problem Exists/Forall SMT solving or EF solving for short. As in general SMT solving, we are not just interested in a yes or no answer. We want to produce a model when the formula is satisfiable. Our actual problem is then to search for a vector of values a for the existential variables, such that $\forall y.\Phi(a,y)$ is true.

EF solving naturally occurs in many synthesis problems. For example, assume we are given a transition system $\mathcal{M}$ and we want to show that a property P is invariant for $\mathcal{M}$. We assume $\mathcal{M}$ is specified by an initialization formula $I(s)$ over state variables S and a transition formula $T(s,s')$ over current and next-state variables S and S'. Showing that P is invariant typically requires finding an auxiliary inductive invariant Q that implies P. In other words, Q must satisfy three constraints:

$$\forall s.I(s) \Rightarrow Q(s) \qquad \forall s,s'.Q(s) \wedge T(s,s') \Rightarrow Q(s') \qquad \forall s.Q(s) \Rightarrow P(s).$$

Automatically constructing such a predicate Q is not trivial. A simplification is to restrict the search space by specifying a *template* for Q. Such a template is a formula of the form $Q(t,s)$ where s denotes the state variables of $\mathcal{M}$ as before, and t is a finite set of parameters. The problem is now to find values for the parameters such that

$$[\forall s.I(s) \Rightarrow Q(t,s)] \wedge [\forall s,s'.Q(t,s) \wedge T(s,s') \Rightarrow Q(t,s')] \wedge [\forall s.Q(t,s) \Rightarrow P(s)].$$

Modulo minor syntactic rewriting, this is an Exists/Forall problem.

Template-based methods have many applications including reverse engineering of hardware [4], program verification [6, 7], hybrid and continuous dynamical systems [14, 15, 2], geometry [5], and program synthesis [13, 12, 9].

We present the algorithm for solving Exists/Forall problems currently implemented in Yices. We discuss some implementation issues and we survey example applications.

```
$i := 0$
$C_0(x) :=$ *Initial constraints on* $x$

Repeat
    find $a_i$ that satisfies $C_i(x)$                    [E-Solver]
        if no $a_i$ is found, return **unsat**

    search for $b_i$ that satisfies $\neg\Phi(a_i, y)$    [F-Solver]
        if no $b_i$ is found, then $a_i$ is a solution;
            return **sat**

    *Generalize from* $b_i$: compute a constraint $G(x)$ such that
        1) $G(a_i)$ is true
        2) $G(x) \Rightarrow (\exists y : \neg\Phi(x, y))$

    $C_{i+1}(x) := C_i(x) \land \neg G(x)$
    $i := i + 1$
end
```

Figure 1: Two-Solver Procedure for the EF Problem $\exists x \forall y \Phi(x, y)$

2 SMT-Based EF Solving

In principle, many EF problems could be solved using quantifier-elimination methods: rewrite $\forall y.\Phi(x, y)$ into a an equivalent quantifier-free formula $\Phi'(x)$ then check whether $\Phi'(x)$ is satisfiable. Quantifier elimination is applicable for theories such as linear arithmetic or when the universal variables have a finite domain. However, quantifier elimination is typically very expensive and requires specialized algorithms. It also does more than we need. The formula $\Phi'(x)$ characterizes all the solutions to the original problem. We need only one.

An alternative is to combine two ordinary SMT solvers for quantifier-free formulas. A first solver (the E-Solver) generates candidate solutions for the existential variables x. Another solver (the F-Solver) checks whether a candidate solution a is correct, by trying to refute the formula $\Phi(a, y)$. The general procedure is sketched in Figure 1:

- Formula $C_i(x)$ encodes the set of potential candidate for the existential variables.
- At each iteration of the loop, the E-solver checks satisfiability of $C_i(x)$. If the formula is not satisfiable, all candidates have been eliminated so the EF problem is unsatisfiable.
- If $C_i(x)$ is satisfiable, then the E-solver produces a candidate a_i. The F-solver checks whether a_i is correct by checking satisfiability of the formula $\neg\Phi(a_i, y)$. If this formula is not satisfiable, then a_i is a solution to the EF problem otherwise the F-Solver produces a counterexample b_i.
- A key part of the algorithm is to generalize this counterexample. Abstractly, a generalization procedure must construct a formula $G(x)$ (using a_i and b_i) that eliminates invalid candidates for the existential variables. As stated in Figure 1, $G(x)$ must satisfy two requirements:
    - $G(a_i)$ is true (so that at least a_i is removed from the set of candidates).
    - $G(x) \Rightarrow (\exists y.\neg\Phi(x, y))$ is valid (no x that satisfies $G(x)$ is a good candidate).

  The set of candidates $C_{i+1}(x)$ is then updated accordingly and the loop repeats.

Section 4 describes how we compute the initial constraints in Yices. This is not as important as the generalization procedure and it is always safe to simply set $C_0(x)$ to true.

For the generalization procedure, a baseline is to use

$$G(x) := (x = a_i),$$

which just removes the current candidate a_i. This basic form is sound but it does not take the counterexample b_i into account. A more effective approach is to remove all candidates for which b_i is a counterexample. We call this *generalization by substitution:*

$$G(x) := \neg\Phi(x, b_i).$$

If the variables x have a finite domain, then the algorithm obviously terminates. It also terminates if the variables y have a finite domain and generalization by substitution is used. In other cases, a more powerful generalization method is required. For example, the following EF problem in linear arithmetic is not satisfiable

$$\exists x.\forall y.x < y,$$

but generalization by substitution does not converge on this input. To solve EF problems in linear real arithmetic, Yices includes a generalization method based on projection. This method is described in the next section and it can be seen as a form of quantifier elimination that is directed by the model. A similar technique was proposed in the context of property-directed reachability by Komuravelli et al. [10].

3 Model-Guided Generalization

Generalization from a model starts with a quantifier-free formula F whose variables are split into two sets $X = \{x_1, \dots, x_n\}$ and $Y = \{y_1, \dots, y_m\}$. To simplify the presentation, we assume in this section that all the variables are real-valued and that F is a linear arithmetic formula. More precisely, we assume that F is a formula in the logic defined by the following grammar:

$$\begin{array}{rcl} t & := & c \mid x_i \mid y_j \mid t + t \mid c.t \\ l & := & (t > 0) \mid (t \leq 0) \mid (t = 0) \\ f & := & l \mid (\neg f) \mid (f \vee f) \mid (f \wedge f) \end{array}$$

where c can be any rational constant. Let $\mathcal{M}$ be a model of F. Our goal is to construct a formula G that contains only the variables X and such that $G \Rightarrow (\exists y_1, \dots, y_m.F)$. In addition, G must be true in $\mathcal{M}$.

We proceed in two steps. First, we construct an implicant B of F based on the model $\mathcal{M}$. Then, we eliminate the Y variables from B, again by taking advantage of the model $\mathcal{M}$.

3.1 Implicant

The implicant B is a conjunction of arithmetic literals that implies F and such that B is true in $\mathcal{M}$. We also ensure that no literal of B is of the form $\neg(t = 0)$. The implicant can be easily constructed by a top-down traversal of the formula F. We can use the following rules and set

$B := \mathit{Imp}^+(F)$.

$$\begin{array}{rcll}
\mathit{Imp}^+(l) & := & l & \\
\mathit{Imp}^+(f_1 \vee f_2) & := & \mathit{Imp}^+(f_1) & \text{if } f_1 \text{ is true in } \mathcal{M} \\
& := & \mathit{Imp}^+(f_2) & \text{otherwise} \\
\mathit{Imp}^+(f_1 \wedge f_2) & := & \mathit{Imp}^+(f_1) \wedge \mathit{Imp}^+(f_2) & \\
\mathit{Imp}^+(\neg f) & := & \mathit{Imp}^-(f) &
\end{array}$$

$$\begin{array}{rcll}
\mathit{Imp}^-(t = 0) & := & (t > 0) & \text{if } t \text{ has a positive value in } \mathcal{M} \\
& := & \neg(t \geq 0) & \text{if } t \text{ has a negative value in } \mathcal{M} \\
\mathit{Imp}^-(t > 0) & := & \neg(t > 0) & \\
\mathit{Imp}^-(t \geq 0) & := & \neg(t \geq 0) & \\
\mathit{Imp}^-(f_1 \vee f_2) & := & \mathit{Imp}^-(f_1) \wedge \mathit{Imp}^-(f_2) & \\
\mathit{Imp}^-(f_1 \wedge f_2) & := & \mathit{Imp}^-(f_1) & \text{if } f_1 \text{ is false in } \mathcal{M} \\
& := & \mathit{Imp}^-(f_2) & \text{otherwise} \\
\mathit{Imp}^-(\neg f) & := & \mathit{Imp}^+(f) &
\end{array}$$

In these rules, $\mathit{Imp}^+(f)$ is an implicant of a formula f if f true in $\mathcal{M}$ and $\mathit{Imp}^-(f)$ is an implicant of $\neg f$ when f is false in $\mathcal{M}$.

3.2 Variable Elimination

After the implicant is computed, we can further normalize the literals of B to replace $\neg(t > 0)$ by $-t \geq 0$ and $\neg(t \geq 0)$ by $-t > 0$. Without loss of generality, we can then assume that B is a conjunction of arithmetic atoms containing the variables X and Y. It remains to eliminate the variables Y from these atoms. One could use Fourier-Motzkin elimination but this is not practical in general as it may cause an exponential blowup in the number of atoms.

A key observation is that we do not need to preserve equivalence. In our application, it is enough to construct a formula G such that

$$G \Rightarrow (\exists y_1, \ldots, y_m . B).$$

As noted in [10], we can compute such a G efficiently by employing ideas related to virtual term substitution [17, 11, 18].

Virtual substitution methods eliminate quantifiers by computing so-called *elimination sets*. Given a formula ϕ and a variable y that is free in ϕ, an elimination set for the variable y is a finite set V of terms that do not contain y and such that

$$(\exists y : \phi) \Leftrightarrow \bigvee_{t \in V} \phi[t/y].$$

Since we do not need to preserve equivalence, it is sufficient in our context to pick a single term t out of V. We then obtain

$$\phi[t/y] \;\Rightarrow\; (\exists y : \phi).$$

We also have a model $\mathcal{M}$ of ϕ at our disposal and the choice of t is guided by this model. We must pick a term t such that $\phi[t/y]$ is true in $\mathcal{M}$.

Virtual term substitution can be complex. For many theories, the substitution set may contain terms—such as infinity, square roots, or infinitesimals—that are not formally in the logic. The *virtual substitution* $\phi[t/y]$ must ensure that such terms do not occur in the resulting formula. In our context, everything is simple and ordinary substitution works fine. The main step in our variable-elimination algorithm removes a single variable y from a conjunction of linear inequalities. For example, consider a conjunction ϕ of the form

$$\bigwedge_j (y > t_j) \;\wedge\; \bigwedge_k (y < u_k)$$

where t_j and u_k are linear-arithmetic terms that do not contain y. We first evaluate all the terms t_j and u_k in the model $\mathcal{M}$. Let t_{j_0} be a term that has maximal value in $\mathcal{M}$ among all t_js and let u_{k_0} be a term that has minimal value in $\mathcal{M}$ among all u_ks, then we use

$$t \quad := \quad \frac{t_{j_0} + u_{k_0}}{2}$$

as the substitution term. The resulting $\phi[t/y]$ is $(\bigwedge_j (t > t_j) \wedge \bigwedge_k (t < u_k))$, which is true in $\mathcal{M}$.

Each variable-elimination step takes a conjunction of arithmetic atoms as input. It picks a universal variable y to eliminate, constructs an elimination term t as sketched previously, then replaces y by t. The substitution does not cause an increase in the number of atoms and the result is a conjunction of arithmetic atoms. We can then iterate this procedure to eliminate all the universal variables one by one.

3.3 Properties

The model-guided quantifier elimination procedure satisfies our requirements. Given an initial formula F and a model $\mathcal{M}$, it constructs a formula G that does not contain the variables Y and such that $G \Rightarrow (\exists y_1, \ldots, y_m.F)$ and G is true in $\mathcal{M}$. The procedure also ensures that G is a conjunction of arithmetic atoms. Furthermore, there are finitely many possible such G's for a given formula F. This follows from the fact that F has finitely many implicants (as we construct them) and that there are finitely many choices for the variable y and term t in each variable-elimination step. Employing this generalization algorithm ensures then that the EF solver loop (Fig. 1) terminates.

4 Implementation Details

We now discuss a few implementation issues.

4.1 Preprocessing

So far, we have considered problems of the form $\exists x. \forall y. \Phi(x, y)$. In practice, Yices works on problems of the form

$$\begin{array}{l} \exists x : A(x) \\ \quad \wedge \quad (\forall y_1 : B_1(y_1) \;\Rightarrow\; D_1(x, y_1)) \\ \qquad\qquad \vdots \\ \quad \wedge \quad (\forall y_k : B_n(y_k) \;\Rightarrow\; D_k(x, y_k)) \end{array}$$

Any EF problem can be written in this format. This amounts to pushing the universal quantifiers inside the formula Φ (this is known as *miniscoping*). It is also important to eliminate variables by substitution whenever possible as explained in [4].

4.2 Initial Constraints on Existential Candidates

To learn initial constraints on the existential variables, a simple approach is to compute a sample of values $b_1, \ldots, b_t$ for the universal variables then substitute these values for y in Φ. This gives $C_0(x) := \Phi(x, b_1) \wedge \ldots \wedge \Phi(x, b_y)$, which helps provided the subformulas $\Phi(x, b_j)$ are not trivially true. To find the samples, Yices takes advantage of formulas of the form $\forall y : B(y) \;\Rightarrow\; D(x, y)$ that may result from preprocessing. Given such a formula, we can search for models of $B(y)$. Each such model gives a sample b_i and we learn an initial constraint $D(x, b_i)$, which is not likely to reduce to true.

4.3 Implicant Construction

We have glossed over the details of implicant computation. In practice, the formulas are not as simple as presented in section 3.1. The implementation supports the full Yices 2 language and can process if-then-else terms and atoms such as $(\texttt{distinct}\ t_1 \ldots t_k)$. For example, an implicant for $1 + (\texttt{ite}\ c\ u\ v) \geq 0$ can be either $c \wedge 1 + u \geq 0$ or $\neg c \wedge 1 + v \geq 0$, depending on whether c is true or false in the model.

It is also useful to preserve equalities between Boolean terms. If t is a Boolean variable, Yices treats the formula $t \;\Leftrightarrow\; u$ as an atomic equality (to enable variable substitution). Otherwise, the formula $t \;\Leftrightarrow\; u$ is treated as the equivalent formula $(t \wedge u) \vee (\neg t \wedge \neg u)$.

4.4 Variable Elimination

The actual variable elimination algorithm implemented in Yices uses a hybrid of virtual-term substitution and Fourier-Motzkin elimination. It first eliminates variables by Gaussian elimination if the implicant B contains arithmetic equalities. After this, all the atoms are linear inequalities. Yices then applies Fourier-Motkzin elimination when it is cheap (i.e., when it does not increase the total number of atoms) in preference to the virtual-substitution method presented in Section 3.2.

5 Example Applications

We survey recent applications of Yices to different classes of EF problems.

5.1 Control and Priority Synthesis

Cheng et al. [2] apply EF-SMT to design and verification problems related to Cyber-physical systems. The software and examples used by Cheng are available at `http://www6.in.tum.de/~chengch/efsmt/`. The implementation used an older version of Yices 2 as a backend SMT solver. Several of the example problems involve non-linear arithmetic constraints over the reals, but Cheng converted them all to finite-precision bitvector arithmetic.

Figure 2 shows one of Cheng's example converted to the Yices syntax. This example synthesizes a Lyapunov function to prove stability of a simple dynamical system. The specifications start with the declaration of two bitvector variables `a` and `r`, which are two coefficients in the Lyapunov function. Thus, `a` and `r` are the existential variables of this problem. The first assertion in Figure 2 gives lower and upper bounds on `a` and `r`. The universal variable `z` occurs under the `forall` quantifier in the second assertion. It denotes the state of the dynamical system. The command `(ef-solve)` checks satisfiability of these constraints using Yices's EF solver then `(show-model)` displays the result. In this case, Yices produces:

```
(define a :: (bitvector 20))
(define r :: (bitvector 20))

(assert
 (and (bv-slt 0b00000000000000000000 a) (bv-slt 0b00000000000000000000 r)
      (bv-slt a 0b00000000110010000000) (bv-slt r 0b00000000110010000000)))

(assert
 (forall (z :: (bitvector 20))
   (=>
    (and (bv-slt 0b00000000000000000000 (bv-add 0b00000000000000000001 r z))
         (bv-slt (bv-add 0b11111111111111111111 (bv-mul 0b11111111111111111111 r) z)
                 0b00000000000000000000)
         (/= z 0b00000000000000000000))
    (and
     (=>
      (bv-sge (bv-mul 0b00000000000001000000 a) 0b00000000000000000000)
      (or
       (bv-sge (bv-add 0b11111111111111111111 z) 0b00000000000000000000)
       (and (bv-sge (bv-add 0b11111111111111111111 z) 0b11111111111111000000)
            (bv-sge 0b00000000000000000000 (bv-add 0b00000000000000000001 z)))))
     (or (bv-sge (bv-mul 0b00000000000001000000 a) 0b00000000000000000000)
         (bv-sge 0b11111111111110100000 (bv-add 0b00000000000000000001 z)))))))

(ef-solve)
(show-model)
```

Figure 2: Example EF Problem from [2] in the Yices Syntax

Run times in ms

Solver	Control				Prio. Synth.				
Cheng	0	0	10	10	20	20	170	2430	6740
Yices	0	0	4	8	4	4	16	96	176

Number of iterations

Solver	Control				Prio. Synth.				
Cheng	2	2	3	7	4	4	18	111	11
Yices	1	1	4	7	1	1	10	68	6

Table 1: Results on Small Problems

```
sat
(= a 0b00000000100000000000)
(= r 0b00000000000000100000)
```

All examples in [2] are relatively small. Four examples are related to control and are encoded in bitvector arithmetic. Five more examples are purely Boolean and correspond to priority synthesis in the BIP framework [1]. Table 1 shows the runtime of Yices on these problems. For comparison, the table also shows the runtime of Cheng's original EF-SMT solver implementation, which was an outer loop making calls to Yices's API. Cheng's outer loop had a significant overhead and the new EF solver in Yices is considerably more efficient.

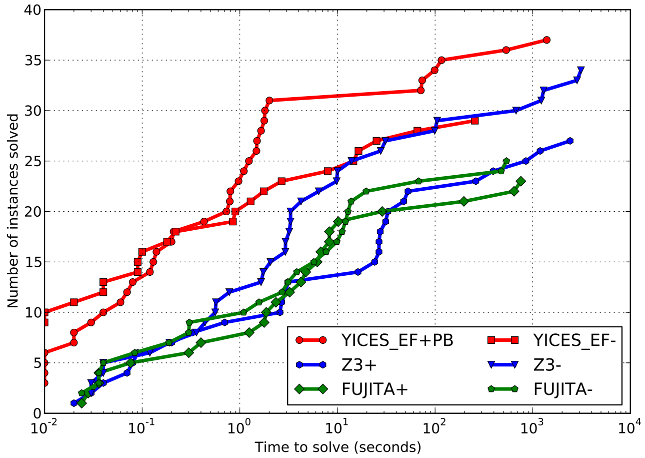

Figure 3: Performance of EF Solvers on Hardware Problems

5.2 Reverse Engineering of Hardware

Gascón et al. [4] use the Yices EF-SMT solver in reverse engineering of hardware. The goal is to identify the functionality of a low-level combinational circuit by searching for a match in a fixed library of known components. For example, we may be given a gate-level circuit description C and attempt to show that it implements an n-bit adder. The main difficulty is that the roles of the input and output ports of C are not known. One must discover how to map these ports to the input or output bits of an adder. In general, the problem is more complex as one may have to treat inputs of C as control signals and show that C behaves like an adder only for some values of these control input.

As shown in [4], this matching problem can be formulated as an EF problem over the Booleans (or bitvectors). The encoding uses a signature-based simplification technique to statically detect that pairs of input signals cannot be matched. This simplification is important

Figure 3—reproduced from [4]—shows empirical results on 39 benchmarks. The graph shows the runtimes and number of benchmarks solved by Yices and two other solvers. These experiments used a timeout of 3600s of CPU time. The problem can be solved using Z3's support for quantified bitvector constraints [19] and with a specialized form of 2QBF solver [8, 3] that is referred to as Fujita in the figure. The positive and negative signs in the figure correspond to two encodings of permutation that differ in the polarity of literals (see [4] for details). The encoding used has a significant impact on the performance of Yices and Z3. We should also point that off-the-shelf QBF solvers did not work well on these benchmarks.

5.3 Program Synthesis

Ashish Tiwari and Adrià Gascón are developing Synudic, a program-synthesis framework that relies on assigning a *dual interpretation* to basic components[1]. This approach is related to program sketching [12]. Synudic includes a language for specifying sketches and basic program

[1] The tool is available at http://www.csl.sri.com/users/tiwari/softwares/auto-crypto/.

components. Each basic component is defined by a signature and is given two interpretations. The functional interpretation defines what the component does and a non-functional interpretation captures additional properties.

For example, in a cryptographic application, one can declare a symbol `oplus` of arity two. The functional interpretation specifies that `oplus` computes the bitwise XOR of its two arguments. The non-functional interpretation captures cryptographic properties of XOR. This abstract non-functional interpretation can be seen as a type system that rules out insecure combinations of components.

The Synudic tool converts the input specification into an Exists/Forall problem and uses Yices 2 as backend solver. Using this method, one can automatically generate straight-line bitvector manipulation programs, padding schemes for public-key encryption, and synthesize block cipher modes of operations. More details on the tools and example applications are presented in a forthcoming paper [16]. A particularly promising feature of Synudic is the possibility of encoding constraints on the solution space using the non-functional interpretation.

6 Limitations and Future Work

Performance of the EF solving algorithm depends crucially on the generalization procedure. It assumes that one can learn useful facts from a counterexample b_i that satisfies $\neg\Phi(a_i, y)$. This hypothesis appear to hold in many applications, but it is easy to find counterexamples. For example, consider the problem

$$\exists x.\forall y.x \neq h(y).$$

For this type of problems, the model-based generalization procedures discussed previously never learn anything useful. A counterexample b_i eliminates only one candidate value a_i for the existential variables because we then have $a_i = h(b_i)$. In such examples, model-based generalization does not do better than our baseline, which consists of enumerating all candidates one by one.

We are currently developing support for non-linear real arithmetic in Yices, which we hope will enable Yices to solve EF problems that are most relevant to analysis and synthesis for dynamic and hybrid systems. We are also planning to extend the model-based generalization procedures to linear integer arithmetic.

7 Conclusion

Exists/Forall solving is a useful functionality that can be implemented using existing SMT solvers for quantifier-free formulas. It has many applications related to synthesis.

Generalizing from models is also a useful function of its own. It can be seen as complementary to interpolant construction, as it learns from a model rather than a proof of unsatisfiability. Both model-based generalization and implicant construction are now part of the Yices API.

References

[1] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogenous real-time components in BIP. In *Software Engineering and Formal Methods (SEFM 2006)*, pages 2–12. IEEE Computer Society Press, 2006.

[2] Chih-Hong Cheng, Natarajan Shankar, Harald Rueß, and Saddek Bensalem. EFSMT: A Logical Framework for Cyber-Physical Systems. `arXiv:1306:3456b2` and `http://www6.in.tum.de/~chengch/efsmt/`, June 2014.

[3] Masahiro Fujita, Satoshi Jo, Shohei Ono, and Takeshi Matsumoto. Partial synthesis through sampling with and without specification. In Jörg Henkel, editor, *The IEEE/ACM International Conference on Computer-Aided Design (ICCAD'13)*, pages 787–794, 2013.

[4] Adrià Gascón, Pramod Subramanian, Bruno Dutertre, Ashish Tiwari, Dejan Jovanović, and Sharad Malik. Template-based circuit understanding. In *Formal Methods in Computer-Aided Design (FMCAD 2014)*, pages 83–90, October 2014. www.cs.utexas.edu/users/hunt/FMCAD/FMCAD14.

[5] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometric constructions. In *PLDI'11: Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–61. ACM, June 2011.

[6] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI'08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 281–292. ACM, 2008.

[7] Sumit Gulwani and Ashish Tiwari. Constraint-based approach for analysis of hybrid systems. In Aarti Gupta and Sharad Malik, editors, *Computer-Aided Verification (CAV'2008)*, volume 5123 of *Lecture Notes in Computer Science*, pages 190–203. Springer, 2008.

[8] Mikolás Janota and João P. Marques Silva. Abstraction-Based Algorithm for 2QBF. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing (SAT 2011)*, volume 6695 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2011.

[9] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 215–224, 2010.

[10] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification (CAV 2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2014.

[11] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *Computer Journal*, 36(5):450–462, 1993.

[12] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5–6):475–495, October 2013.

[13] Armando Solar-Lezama, Rodric Rabbah, Ratislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 281–294. ACM, June 2005.

[14] Thomas Sturm and Ashish Tiwari. Verification and synthesis using real quantifier elimination. In *Proceedings of the 36th International Symposium on Symbolic and Algebraic Computation*, pages 329–336. ACM, 2011.

[15] Ankur Taly, Sumit Gulwani, and Ashish Tiwari. Synthesizing Switching Logic using Constraint Solving. *International Journal on Software Tools for Technology Transfer*, 12(6):519–535, November 2011.

[16] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. Program synthesis using dual interpretation. In *25th International Conference on Automated Deduction (CADE 2015)*, 2015. to appear.

[17] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1–2):2–27, February–April 1988.

[18] Volker Weispfenning. Quantifier eliminatio for real algebra — the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2):85–101, January 1997.

[19] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.