Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Template-based Program Verification and Program Synthesis

Saurabh Srivastava[⋆] and Sumit Gulwani[⋆⋆] and Jeffrey S. Foster[⋆⋆⋆]

University of California, Berkeley and Microsoft Research, Redmond and University of Maryland, College Park

Received: date / Revised version: date

Abstract. Program verification is the task of automatically generating proofs for a program's compliance with a given specification. Program synthesis is the task of automatically generating a program that meets a given specification. Both program verification and program synthesis can be viewed as *search problems*, for proofs and programs, respectively.

For these search problems, we present approaches based on user-provided insights in the form of *templates*. Templates are hints about the syntactic forms of the invariants and programs, and help guide the search for solutions. We show how to reduce the template-based search problem to satisfiability solving, which permits the use of off-the-shelf solvers to efficiently explore the search space. Template-based approaches have allowed us to verify and synthesize programs outside the abilities of previous verifiers and synthesizers. Our approach can verify and synthesize difficult algorithmic textbook programs (e.g., sorting, and dynamic programming-based algorithms, etc.), and difficult arithmetic programs.

1 Introduction

Programming is a difficult task. The programmer comes up with the high-level insight of the algorithm to be applied. They then conjecture a particular sequence of operations to be performed. While these first two conceptualizing tasks require programmer insight, they are relatively less time consuming and less mundane than the subsequent step of writing the program in C, C++, Java etc. There is additional effort in ensuring that all corner cases are covered, which is a tiresome and manual process. The dilligent programmer will also verify that the final program is not only correct but also mechanically verifiable, which usually requires them to add annotations illustrating the correctness.

This makes programming a subtle task, and the effort involved in writing correct programs is so high that in practice we accept that programs can only be approximately correct. While developers do realize the presence of corner cases, the difficulty inherent in enumerating and reasoning about them, or using a verification framework, ensures that developers typically leave most of them to be uncovered during the subsequent fixes of the code. This scenario ensures that even the manual task of coding the algorithm in a programming language requires skill, and so we trust the more skillful programmers to make fewer mistakes, i.e., envision and cover more corner cases. Since in common practice there is no formal system to ensure correctness, we have a scenario where an inherently engineering task is now more of an art. While the step of coming up with the algorithmic insight necessarily involves human intuition, we believe translating that into a correct program should be fairly mechanical. The correctness of the program written should not depend on the programmers skill but should instead be an outcome of the programming system.

To this end, we propose *template-based programming*. We propose a system for *program verification*, which proves programs correct, and for *program synthesis*, which automatically generates programs, using templates. We will employ proof-templates for verification, and proof- and program-templates for synthesis. Templates are a mechanism for the user to provide the system with their high-level insight about the desired program. The system finds a proof within a template, when verifying a

⋆ e-mail: saurabhs@cs.berkeley.edu. This material is based upon work supported by the National Science Foundation under Grant #1019343 to the Computing Research Association for the CIFellows Project.

⋆⋆ e-mail: sumitg@microsoft.com

⋆⋆⋆ e-mail: jfoster@cs.umd.edu

given program. Or it finds a program within a template, when synthesizing a program to match a given specification. Notice that our proposal is orthogonal to the programming discipline of generic programming or template meta-programming found in languages such as C++. In our approach the template is a incomplete scaffold of the final program or proof, and has no meaning until the synthesizer or verifier has filled in the incomplete holes.

We believe template-based programming is a significantly better approach to program development. Algorithmic insights and design will still require artful human insight, but template-based program verification and program synthesis reduce the art in software coding. Program verification helps ensure that all corner cases are covered, if the programmer manually writes the program, and is willing to provide the insights about the proof using templates. Program synthesis not only covers corner cases (by generating verified programs), it virtually eliminates the manual coding aspect, by limiting programmer input to insights about the program structure (which are very close to the algorithmic insights they already have in mind), in the form of templates.

In this paper, we describe work we have been pursuing in the domain of template-based programming. In the rest of the paper, we will give a broad overview of the ideas behind template-based program verification [29, 57, 30], and program synthesis [59]. We implemented these ideas in the VS^3 toolset [58], which provides three tools, that take as input proof- and program-templates as described in Section 2. Templates and programs are interpreted as formulae as described in Section 3. Using these interpretations, the tool $\texttt{VS}^3_{\texttt{LIA}}$ infers invariants over linear arithmetic, the tool $\texttt{VS}^3_{\texttt{PA}}$ infers invariants over predicate abstraction, and the tool $\texttt{VS}^3_{\texttt{SYN}}$ synthesizes programs, as described in Section 4.

2 Programs, Correctness Assertions, Invariant Templates, and Program Templates

Programs for us will be standard imperative programs in a language like C. For instance, the following might be an example program:

```
TwoPhase(int n) {
1  Assume n ≥ 0;
2  x := 0; y := 0;
3  while (y ≥ 0) {
4     if (x ≤ n) y++;
5     else y--;
6     x++;
7  }
8  Assert relates(x, n);
}
```

Notice the use of `Assume` and `Assert` statements. `Assume` allows the programmer to tell the system that whenever control reaches the `Assume`, its argument, a boolean function, evaluates to *true*. `Assert` allows the programmer to state requirements that the program should meet. So in this case, the programmer stated that the function `TwoPhase` will always be called with an n greater than 0. Additionally, they want the program to satisfy a `relates` clause between x and n. That clause may be the following:

```
bool relates(int x, int n) {
1  return x == 2n + 2;
}
```

These are *correctness assertions* for the program. In practice, the C library implementation of `assert` ensures the correctness assertions by evaluating its boolean function argument and if it evaluates to $false$ then aborting execution. Such runtime failure is not desirable, hence programmers typically try to test the program over a couple of inputs and ensure that the assertions are satisfied. But such testing does not guarantee correctness over arbitrary inputs to the program. Verification attempts to provide such guarantees. Program verification is done using properties that hold at a program point whenever control reaches that program point. Since these properties hold at all times control reaches that location, they are *invariant*, and called as such. For instance, two facts that holds in between the two assignments on line 2 are $x = 0$ and $n \geq 0$, thus an invariant that holds there is $x = 0 \wedge n \geq 0$. It turns out the important invariants needed for verifying a program correct are the ones right at the start of loops, i.e., at loop headers, for example on line 3 in our program. One may look at the start of the program and incorrectly conclude that $x = 0 \wedge y = 0 \wedge n \geq 0$ is invariant. This invariant is too strong. In particular, it is violated when control traverses the loop and reaches back to line 3. Another invariant, and one that holds across loop iterations, would be $x \geq 0 \wedge n \geq 0$ because it is maintained across loop iterations. But it does not serve our purpose, which is to prove the program correct, given the correctness assertion. In fact, the real invariant required to show the assertion correct is: $(0 \leq x \leq n+1 \wedge x = y) \vee (x \geq n+1 \wedge y \geq -1 \wedge x+y = 2n+2)$, or written more programmatically:

```
bool invariant(int x, int y, int n) {
1  bool fact1 := (0 ≤ x) ∧ (x ≤ n + 1) ∧ (x = y);
2  bool fact2 := (x ≥ n + 1) ∧ (y ≥ −1) ∧ (x + y = 2n + 2);
3  return fact1 ∨ fact2;
}
```

Even in this more amenable form, we have immediately gotten to the point where the overhead to the programmer is very severe. Proving this small program correct requires too much annotation effort, even though we have not even discussed why exactly the invariant contains these many terms. It is unlikely that a programmer will write this invariant by hand. This motivates the use of *invariant templates*. An invariant template is exactly like our `invariant` function above, except that we will only leave the trivial parts intact and delete the rest.

```
  bool template(int x, int y, int n) {
1   bool fact1 := [-] ∧ [-] ∧ [-] ∧ [-];
2   bool fact2 := [-] ∧ [-] ∧ [-] ∧ [-];
3   return fact1 ∨ fact2;
  }
```

where the programmer has left out holes "`[-]`", which the tool will fill out with appropriate facts, such that the invariant proves the correctness assertions, assuming that such an invariant exists. The invariant may not exist if the program is faulty, i.e., there exists some input over which the correctness assertion is violated. Another important thing to note about the invariant template is that is it not entirely devoid of information. It does impose a certain structure, namely a disjunction (of size 2) of conjunctions (of size 4 each). This captures the programmer's insight about what the proof about the program would look like. Examining `TwoPhase` we may notice that there are two phases, one in which y is increasing, and another in which it is decreasing. Hence the disjuncts. We may also guess, that we may need facts concerning x, y and one relating them, hence at least three, although that is more tenuous, and maybe an educated guess suffices. Note also that a template does not need to be exact, and may be more general than needed. For instance, `template` contains four conjuncts in each disjunct, while `invariant` only needs three. On the other hand, a template that is too specific, i.e., not general enough, will not suffice. Verification will fail on too specific a template, similar to how $x \geq 0 \wedge n \geq 0$ did not suffice.

When writing programs manually programmers typically are comfortable writing correctness assertions such as `relates`. They mostly forgo correctness verification because of the effort involved in providing details such as in `invariant`. Template-based verification on the other hand requires only the insight provided as hints such as `template` while the tool fills out the details of the proof, i.e., constructs `invariant` from `template`, thus bringing the overhead low enough to be used in practice. We will encounter more sophisticated invariant templates later, but in all we will note a significant reduction in programmer effort as compared to writing the invariants manually.

We can apply the same principle of omitting the details using template holes for program statements and guards, attempting the more ambitious task of generating the program from the insights provided by the user. To be more specific, instead of the programmer writing a program, they can write a *program template* which specifies some program structure, e.g., where the loops appear, while leaving out the details such as guards, conditionals, and assignments, as much as required.

For instance, if the programmer was unsure of the body of the loop in our example, they may write the following program template instead:

```
  TwoPhaseTemplate(int n) {
1   Assume n ≥ 0;
2   x := 0; y := 0;
3   while (y ≥ 0)
4       [-]→ [-]_y
5       [-]→ [-]_y
6       x++;
7   }
8   Assert relates(x, n);
  }
```

where the block consisting of two `[-]` $\rightarrow$ `[-]`$_y$ indicates that there is a guard hole for a predicate, the first "`[-]`", which if satisfied triggers the statements in the statement hole, the second "`[-]`$_y$" with the subscript denoting that the hole needs to be instantiated with assignment to y. This example hints at how program templates allow the programmer to encode the intuition behind the algorithm, without manually writing down the expressions of the program. The synthesizer takes care of instantiating the holes such that they meet the correctness assertions. In more complicated synthesis tasks we will see later, the correctness assertions encode the entire specification of the task (modulo termination). Imposing structure using the program template, and correspondingly the invariant templates, limits the synthesizer to programs that are semantically correct with respect to the specification. This allows the tool to generate verified programs given just the correctness assertions and program template.

The informed reader will notice similarities between our program templates and Sketches in the SKETCH tool [55]. While similar, there are subtle but important differences. The only structure that is necessarily required for program templates are the location of loops, while in Sketches all control flow, and which assignments occur in which order, are part of the structure that needs to be specified. Additionally, Sketch synthesizes integers that are bounded for each hole, while program templates contain holes that are arbitrary expressions and predicates, and therefore the synthesizer needs to generate values that are potentially unbounded. Additionally, and perhaps most importantly, Sketch only has the notion of partial programs, while our system additionally allows partial invariants to be specified as well.

2.1 The Origin of Templates

Templates serve two important purposes. First, they are an expressive yet accessible mechanism for the programmer to specify their insight. Second, they limit the search space of invariants and proofs, and consequently make verification and synthesis feasible for difficult cases. We discuss both these in detail below.

First, templates let the programmer specify the intuition behind the program, or proof, in a very systematic format. Any attempt at formalizing intuition will be lacking in significant ways, and so is ours, but we believe our way of using invariant and program templates

is an interesting design point. For any formula or program that we do not know enough about, and want to abstract, using template holes as abstract replacement is a very expressive mechanism. Additionally, it is also very usable, with a minimal learning curve, and because it follows the top to bottom thought process that we believe programmers follow in practice. This belief is substantiated by the observation that it is standard practice in software engineering to first do the high-level design, and then subsequently to fill in components in the design. Templates provide a similar workflow. The programmer specifies the high-level design of the invariants, or program, and the system fills out the details.

Second, templates make verification and synthesis tractable. In their full generality, verification and synthesis are both undecidable problems. All existing verification and synthesis systems make assumptions, typically about the domain over which the program is written [55,36], or needs to be analyzed [9], that sufficiently simplify the problem such that it is amenable to solving. Our assumption is that programmers feel comfortable about partial programs, since techniques like modular programming and libraries after all simply allow the programmer to delegate functionality to an external component, that they do not care about. Similarly, templates limit the programmer's task to the high-level structure of the invariants and code. Under the assumption that templates are provided the verification and synthesis tasks essentially reduce to search problems for the missing components in the template. Search problems are encountered in various subdisciplines of computing, and hence we have various solutions, heuristic or otherwise that we can leverage. In particular, we will leverage satisfiability solving as our mechanism for searching for the satisfying solutions to the holes.

2.1.1 Invariant Templates for Verification

To make this discussion about templates more concrete, let us consider the binary search program:

```
BinarySearch(Array A, int e, int n) {
1   low := 0; high := n − 1;
2   while (low ≤ high) {
3       mid := ⌈(low + high)/2⌉;
4       if (A[mid] < e)
5           low := mid + 1;
6       else if (A[mid] > e)
7           high := mid − 1;
8       else return true;
9   }
10  Assert notContains(A, n, e);
11  return false;
}
```

For this program, we want to prove that when it fails to find the element e in array A of size n, then the element truely did not exist, i.e., more formally that $(\forall j : (0 \leq j < n) \Rightarrow A[j] \neq e)$ holds of the array. This is specified as the `notContains` clause:

```
notContains(Array A, int n, int e) {
1   fact = true;
2   for(j = 0; j < n; j++)   // ... foreach ({j | 0 ≤ j < n})
3       fact = fact ∧ A[j] ≠ e
4   return fact;
}
```

Assuming independent loop iterations, a loop such as the one on line 2, can be written as a set comprehension over $0 \leq j < n$, i.e., as that specified by the comment.

Programmers routinely write programs like `BinarySearch` and correctness assertions like `notContains`. They do not go the extra step to attempt to prove their assertions before running their program (which causes runtime failure if the assertions are violated, due to unanticipated inputs.) Templates facilitate this extra step of verification almost without extra effort.

We now discuss the process of obtaining an invariant template. First, our experience has shown that most invariants need a few simple quantifier-free relations, especially to show inductiveness, i.e., when we traverse a loop and come back to the header. Therefore, we start by adding a simple hole "`[-]`". Then, since we are attempting to prove `notContains`, we should have part of the invariant template be at least as expressive. In particular, it should contain a template for each part:

```
1 fact2 = true;
2 foreach ({j | [-]})
3     fact2 = fact2 ∧ [-]
```

where the iteration is over values of the index variable j whose value space is to be discovered. If we attempt verification with just a simple template hole and one `foreach` as above, it fails. Since the quantifier-free hole is already present, the only option available to the programmer is to add quantified fact. We follow a strategy of adding simpler facts, i.e., with one quantified variable, before increasing expressivity, e.g., by adding facts with nested quantification. Thus, we make the template more expressive and we add a few more `foreach`s:

```
templateBS(..) {
1   fact1 = [-];
2   fact2 = true; foreach ({j | [-]})
3                     { fact2 = fact2 ∧ [-]}
4   fact3 = true; foreach ({j | [-]})
5                     { fact3 = fact3 ∧ [-]}
6   fact4 = true; foreach ({j | [-]})
7                     { fact4 = fact4 ∧ [-]}
8   return fact1 ∧ fact2 ∧ fact3 ∧ fact4;
}
```

Not only is the tool able to verify the program with this template, it informs us of a condition we missed. It tells us that the program can be shown valid, by the invariant:

$$\begin{pmatrix} 0 \leq low \wedge high < n \\ \forall j : (low \leq j \leq high) \Rightarrow A[j] \leq A[j+1] \\ \forall j : (0 \leq j < low) \Rightarrow A[j] \neq e \\ \forall j : (high < j < n) \Rightarrow A[j] \neq e \end{pmatrix}$$

Notice how this is an instantiation of `templateBS`, just written mathematically. (We will discuss later how the

programmer provides hints that enable the tool to get candidate facts, e.g., $0 \leq low$, which it uses to instantiate the holes.) Probably more notably, the tool outputs that the invariant proves `BinarySearch`, but only if it is called with a sorted array, i.e., there is a *(maximally-weak) precondition*:

$$\forall j : (0 \leq j < n) \Rightarrow A[j] \leq A[j+1]$$

Throughout this paper, maximally-weak preconditions refer to maximally-weak assertions of all instantiations of the template, that are valid preconditions. Notice that the precondition is also an instantiation of the invariant template, which is what the tool used as a template for the precondition. Such gains are common when applying a verification tool. Very often there are facts that the programmer assumes always hold, but are never written down. Using a verification tool makes those implicit coding assumptions explicit making the software more maintainable.

Notice how the precondition generated mechanically is readable and sensible as well. This is a side-effect of constraining maximal-weakness. Ensuring maximal-weakness constrains the solver to a precondition that does not have redundant facts. Additionally, by having the programmer specify the invariant template and instantiating the precondition from the template, the human insight about the program structure that is encoded in the inference process.

To prove that a program meets its specification, our tool expects the user to provide just templates, instead of full invariants. The programmer can choose not to use program verification, but if they want verified programs, the use of templates significantly reduces the effort involved as the alternative is to write down the invariants. Experience in program analysis has shown that programmer interaction is very valuable. Verification tools that attempt to be fully automated work well on a narrow class of programs, but when they fail, they fail in ways that makes it difficult to proceed without modifying the underlying tool/theory. Templates provide an expressive mechanism for the programmer to suggest to the tool the expected form of the invariants, allowing expressive invariant inference, without putting the burden of details on the programmer.

2.1.2 Program Templates for Synthesis

In synthesis the tool instantiates a programmer-specified program template to a program matching a specification. A program template is the medium through which the programmer conveys to the tool the insight behind the program. Consider for example the task of drawing a pixelated line, approximating a real line, between points $(0,0)$ and (X,Y). Figure 1 shows the pixelated line in green, approximating the real line in bold. We assume that we need a line in the NE quadrant, i.e., with $0 < Y \leq X$. Our program is expected to generate (x,y) values for the pixelated line. The specification for this task is simply that the output values should not deviate more than half a pixel away from the real value. That is, we need $\mid y - (Y/X)x \mid \leq 1/2$ for any (x,y) the program outputs, or written as our correctness assertion, we need `notFar` to hold:

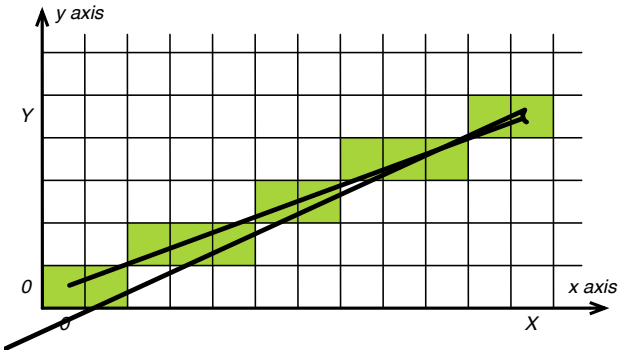

Fig. 1. Drawing a pixelated line approximating the real line from $(0,0)$ to (X,Y) in the NE quadrant, i.e., with $0 < Y \leq X$.

```
bool notFar(int x,y) {
 return −1/2 ≤ y − (Y/X)x ≤ 1/2
}
```

A programmer informed in computer graphics may recognize the specification for which a solution exists using only linear operations. Notice that the program is trivial if we are allowed to divide and multiply, as it would be almost identical to the specification. The challenge is to come up with a program which uses only linear operations, i.e., using only additions and subtractions, as it would be more efficient. If up for a challenge, the reader should stop reading for a moment and attempt to manually write this program. The task is not trivial.

Alternatively, one can use program synthesis! To use template-based synthesis, we need a program template. The program will necessarily loop over some values, and possibly have an initialization preceeding the loop. Thus a plausible program template is:

```
BresenhamsTemplate(int X,Y) {
1  Assume 0 < Y ≤ X
2  [-]→ [-]v1,x,y;
3  while ([-]) {
4     Assert notFar(x,y); // print (x,y)
5     [-]→ [-]v1,x,y
6     [-]→ [-]v1,x,y
7  }
}
```

While it is obvious that the statements will involve updates to x and y, we see an auxiliary variable v_1 being used too. It turns out that when we attempted synthesis without it, the tool failed to generate a program. Therefore we conjectured extra temporaries might be needed taken from the set $\{v_1, v_2, \ldots\}$. Adding a single extra variable v_1 leads to a solution, and we realize that the extra variable is used by the program as a measure of the error variable between the discrete line and the real line.

In fact, a non-trivial solution generated by the synthesis tool is the following:

```
Bresenhams(int X,Y) {
 Assert 0 < Y ≤ X
 v1:=2Y−X; y:=0; x:=0;
 while (x ≤ X)
    if (v1 < 0)
       v1:=v1+2Y;
    else
       v1:=v1+2(Y-X); y++;
    x++;
}
```

Notice how the extra variable plays a critical role in the program, and the programmer did not have to come up with the complicated manipulations to update it and the running coordinates.

Our synthesizer builds on top of verifiers, and consequently as a side-effect of successfully solving for the program, solves for the proof that the program meets the specification. That is, it generates both invariants, for partial correctness, and ranking functions, for termination. In this case, it generates the following invariant:

$$\begin{pmatrix} 0 < Y \leq X \\ v_1 = 2(x+1)Y - (2y+1)X \\ 2(Y-X) \leq v_1 \leq 2Y \end{pmatrix}$$

and the following ranking function:

$$X - x$$

A ranking function is associated with a loop, like an invariant, except for stating a boolean property that holds in each iteration, it is an integer expression that decreases from one iteration to the next. It is also bounded below during iterations of the loop. Thus in this case, we can manually check the validity of the ranking function because $X - x$ is bounded below by 0 and decreases by 1 in each iteration.

If the reader did indeed stop earlier to manually write the program themselves they may want to consider the effort they spent against that involved in writing a program template. There is indeed some trial and error involved, but the actual inference of the program can now be left to the synthesizer. Additionally, they may also consider how long it would take them, or did take them, to find the invariants that were output by the synthesizer as a side effect of template-based program synthesis using verifiers.

Let us revisit the question of whether templates improve efficiency in the context of synthesis. In verification, template specification were for artifacts, i.e., invariant, required for the process of verification. They are of little interest to the programmer. The average programmer is typically only interested in the binary result of whether their program is correct or not; possibly with the exception of missing preconditions under which it may be correct. In synthesis, rather than a template for an artifact, the template is of the unknown program the programmer is expecting to generate. So instead of writing the code from scratch the programmer significantly benefits because they only need to write the structure of the code, and the tool fills out the details.

3 Programs as Formulae

Up until now we have described the programmer input (programs, assertions, and templates), used by our synthesizer and verifier. While our programmatic presentation until now is human readable, the synthesizer and verifier need to translate them into a more amenable form they can reason about. In this section, we describe how programs are interpreted as formulae, using well understood approaches. The novel twist to our choice of formalism will be its ability to handle template holes. This requirement only appears because of our particular template-based approach to verification and synthesis.

3.1 With Invariants, Programs are Just Constraints

The control flow graph of a *structured program*, i.e., one without gotos, has a very specific form. It is acyclic, except for the few backedges created by loops. Using a standard trick from program analysis we can reason about the program using a finite representation, even though actual executions may be infinite. We have already discussed attaching an invariant fact at each loop header, i.e., at the end point of each backedge. If we treat these points as cutting the graph into fragments, then notice that the graph contains no cycles thereafter. The remainder would be program fragments that are acyclic and start at the program entry or an invariant, and end at the program exit or an invariant. Our intention would be to translate each fragment to a formula encoding the fragment's meaning, and thus generate a finite formula that has the meaning of the program. Let us look at the following example, and let us call the (unknown) invariants for the loops on lines 2 and 4, as τ_2^{is} and τ_4^{is}, respectively, named as the invariants for insertion sort at the corresponding lines:

```
   InsertionSort(Array A, int n) {
1    i := 1;
2    while (i < n) {
3       j := i − 1; val := A[i];
4       while (j ≥ 0 ∧ A[j] > val) {
5          A[j + 1] := A[j]; j--;
6       }
7       A[j + 1] := val; i++;
8    }
9    Assert preserves(A, Ã, n);
   }
```

where `preserves` states that the array A has the same contents as an old version, stored in $\tilde{A}$, up to n elements. We will look at the definition of `preserves` later. For now, let us use our trick of starting and stopping at invariants to fragment the program. We reason with the

following finite set (informally written), even though actual runs of the program may be arbitrarily long:

$$
\begin{aligned}
(\mathit{entry} \wedge \mathit{line}\ 1) &\text{ should imply } \tau_2^{is}\\
(\tau_2^{is} \wedge i < n \wedge \mathit{line}\ 3) &\text{ should imply } \tau_4^{is}\\
(\tau_2^{is} \wedge i \geq n) &\text{ should imply } \texttt{preserves}(A, \tilde{A}, n)\\
(\tau_4^{is} \wedge (j \geq 0 \wedge A[j] > \mathit{val}) \wedge \mathit{line}\ 5) &\text{ should imply } \tau_4^{is}\\
(\tau_4^{is} \wedge \neg(j \geq 0 \wedge A[j] > \mathit{val}) \wedge \mathit{line}\ 7) &\text{ should imply } \tau_2^{is}
\end{aligned}
$$

There is no assumption at program entry, so *entry* is *true*. That only leaves the encoding of statements on lines 1, 3, 5 and 7. Looking ahead, in program templates for synthesis statements might have template holes, and so we should ensure that our encoding works for them. The notion of *transition systems* works for our purposes. A transition system is an encoding of a basic block triggered by an optional guard condition. A basic block is encoded as a relation between its outputs and inputs, which are respectively the values of the variables at exit and entry from the block. The outputs values of variables can be indicated by priming the variable name. So the basic block on line 3 is encoded as $j' = i - 1 \wedge \mathit{val}' = \texttt{sel}(A, i)$. When reasoning formally about state update to arrays, we employ McCarthy's theory of arrays [46], which has predicates for array reads $\texttt{sel}(\mathit{array}, \mathit{index})$ and array updates $\texttt{upd}(\mathit{array}, \mathit{index}, \mathit{value})$. The update is non-destructive in that the original array is not modified, but the predicate represents the new array with the *index* location holding *value* instead of the one in *array*.

Using transition systems for reasoning about program fragments works without change in the presence of program templates that contain holes `[-]`, i.e., a placeholder for some expression or predicate that will be filled in later by the synthesizer/verifier. For instance, an assignment $j :=$ `[-]` translates to $j' =$ `[-]`. (The informed reader will notice that the alternative of using Hoare's backwards substitution for reasoning about assignment will not work in the case of template holes.)

Using the above approach, the informal translation we wrote earlier is formally written as *logical constraints*:

$$
\begin{aligned}
(\mathit{true} \wedge i' = 1) &\Rightarrow \tau_2^{is\prime}\\
(\tau_2^{is} \wedge i < n \wedge j' = i - 1 \wedge \mathit{val}' = \texttt{sel}(A, i)) &\Rightarrow \tau_4^{is\prime}\\
(\tau_2^{is} \wedge i \geq n) &\Rightarrow \texttt{preserves}(A, \tilde{A}, n)\\
\begin{pmatrix} \tau_4^{is} \wedge (j \geq 0 \wedge \texttt{sel}(A, j) > \mathit{val}) \\ \wedge\ A' = \texttt{upd}(A, j+1, j) \wedge j' = j - 1 \end{pmatrix} &\Rightarrow \tau_4^{is\prime}\\
\begin{pmatrix} \tau_4^{is} \wedge \neg(j \geq 0 \wedge \texttt{sel}(A, j) > \mathit{val}) \\ \wedge\ A' = \texttt{upd}(A, j+1, \mathit{val}) \wedge i' = i + 1 \end{pmatrix} &\Rightarrow \tau_2^{is\prime}
\end{aligned}
$$

Invariants τ_2^{is} and τ_4^{is} are unknown facts, we cannot directly ask for whether this constraint system is satisfiable. We need to find values for these invariants such that the constraint system is satisfiable. Specifically, τ_2^{is} and τ_4^{is} are placeholder variables for the true invariant assertions, and their primed versions are delayed substitutions. Next we describe the process of discovering these invariants using templates.

3.2 Using Templates in Constraints; Translating to Formulae

The translation of programs to formulae in the previous section contains invariants as symbols, which means we cannot directly check for its consistency. We employ invariant templates to refine the formulae such that it is simpler to solve. But first, let us discuss in more detail what `preserves` is intended to check. The imperatively written `InsertionSort` moves the elements of the array, and not just by operations that preserve elements but by overwriting the array intelligently. (An operation that preserves the element trivially would be swapping.) It is not immediately obvious, neither intuitively and definitely not formally, that the output array contains the same elements as the input. This example shows that correctness assertions can be much more detailed in cases. A first order correctness for a sorting program is that it outputs a sorted array. But a sorted array could be one that contains n replicas of the first element. Having the same set of elements is another aspect of the correctness specification. We examine one part which is of checking that the output array *A contains every element* that was in the input array (lets call it $\tilde{A}$), i.e., $\forall y \exists x : (0 \leq y < n) \Rightarrow (\tilde{A}[y] = A[x] \wedge 0 \leq x < n)$. This does not entirely prove that the elements are the same, as some may be replicated. If we were to prove full correctness, then we would additionally show that the count of elements is the same.

It is easily seen that the following programmatic version of `preserves` encodes the required $\forall\exists$ formula:

```
bool preserves(Array A, Array Ã, int n) {
1  holds = true;
2  // foreach ({y | 0 ≤ y < n})
3  for (y = 0; y < n; y++)
4     found = false;
5     // foreach ({x | 0 ≤ x < n})
6     for (x = 0; x < n; x++)
7        found = found ∨ Ã[y] = A[x];
8     holds = holds ∧ found;
9  return holds;
   }
```

This illustrates how one may write a function that can be matched against a simple syntactic pattern that corresponds to simple quantified formulae.

3.2.1 Expressivity of Templates: General (Quantified) but not Too General (Not Turing Complete)

While writing templates programmatically is easier for the programmer, the solver will need to translate the template back into a form amenable to formal reasoning. Therefore, we have to be careful about the form of the template, and what they encode. For instance, for the above `preserves` property, we conjecture that a template which is similarly expressive would suffice:

```
  template(..) {
1   fact1 = [-];
2   fact2 = true; foreach ({y | [-]})
3                     { fact2 = fact2 ∧ [-]}
4   fact3 = true; foreach ({y | [-]}) {
5                     found = false;
6                     foreach ({x | [-]})
7                        { found = found ∨ [-]};
8                     fact3 = fact2 ∧ found }
9   return fact1 ∧ fact2 ∧ fact3;
  }
```

Notice that this template encodes $\texttt{[-]} \wedge (\forall y : \texttt{[-]} \Rightarrow \texttt{[-]}) \wedge (\forall y \exists x : \texttt{[-]} \Rightarrow \texttt{[-]})$, which is what the tool will syntactically match `template` to. There are certain code patterns that we recognize as being translatable to formulae for templates, in particular right now we recognize patterns for single universal quantified formulae or double alternate universal and existential formulae. The programmer only needs to understand the underlying pattern matching if their template function fails to match any of our patterns. We restrict our attention to simple pattern matches because otherwise inferring the formal description from the template will be another program analysis problem, that we would not want to get into.

If we present this to the tool it infers two invariants, one for each loop, which are each instantiations of the template. The invariant inferred for the outer loop is:

$$\begin{pmatrix} \forall y : (i \leq y < n) \Rightarrow (\tilde{A}[y] = A[y]) \ \wedge \\ \forall y \exists x : (0 \leq y < i) \Rightarrow (\tilde{A}[y] = A[x] \wedge 0 \leq x < i) \end{pmatrix}$$

and the invariant inferred for the inner loop is:

$$\begin{pmatrix} \texttt{val} = \tilde{A}[i] \wedge -1 \leq j < i \ \wedge \\ \forall y : (i < y < n) \Rightarrow \tilde{A}[y] = A[y] \ \wedge \\ \forall y \exists x : (0 \leq y < i) \Rightarrow (\tilde{A}[y] = A[x] \wedge 0 \leq x \leq i \wedge x \neq j+1) \end{pmatrix}$$

4 Solving Technology for Formulae

Until now, we have talked about how one can describe programs, assertions, and invariants, both exactly, and as templates. In this framework, verification and synthesis just reduce to finding a *substitution map for holes* such that the assertions are provably met, when the template is instantiated using the substitution map. In this section, we describe the technology behind finding such opportune substitution maps.

Our translation from the previous section converts a program P to a corresponding formula F. This translation is well known, but mostly in the context of known programs and known (i.e., programmer-specified) invariants. If all the components are known then checking the correctness of P is equivalent to checking whether the formula F holds. More precisely, what we check is that F holds for all possible values of X, the set of program variables that appear in it, i.e., whether $\forall X : F$ holds. But our formulae have holes, for parts of the invariants, and for expressions and predicates if a program template is used instead of a known program. Thus we do not have as simple a checking problem, but instead the problem of finding appropriate values for the holes `[-]` such that when F is instantiated with them it holds. More precisely, we need to find if $\exists \texttt{[-]} \forall X : F$ holds, and if it does, then output the values instantiating the $\exists \texttt{[-]}$.

This discussion immediately points to the possibility of using satisfiability solving for our verification and synthesis problems. SAT solving, and its more expressive version SMT solving, i.e., SAT Modulo Theories, are well studied problems with practical tools that exist now. SAT takes a formulae f with boolean variables and connectives and finds an assignment of the booleans to truth values such that the entire formula evaluates to *true*, i.e., it solves $\exists f$. SMT takes a formula $\bar{f}$ with variables that can more expressive than booleans, such as integers, bit vectors, or arrays, and operators over them, and finds an assignment such that the formula evaluates to *true*, i.e., it solves $\exists \bar{f}$. SMT trivially subsumes SAT. Since SAT is well-known to be NP-Complete, SMT is too, yet heuristic solutions that work well in practice have yielded tools with enormous success in solving instances generated from even industrial applications.

There will be two issues we need to address before we can use SAT/SMT for solving our formulae. First, our formulae contain an additional universal quantification (over program variables X). We will show how we convert to a purely existentially-quantified formula. Second, our holes are more expressive then just integers, arrays, or bit-vectors. They are full expressions, or predicates. Use of templates will come to our rescue here.

4.1 Solving under Domain Assumptions

One way of overcoming the above challenges is by making assumptions about the domains. In particular, we will show how we overcome both these first for the domain of linear arithmetic and then for the domain of predicate abstraction.

4.1.1 Linear Arithmetic

For linear arithmetic, we assume that the program only manipulates linear expressions, and predicates and invariants only refer to linear relations.

Encoding expressions and predicates using integers Under linear arithmetic, we encode program expressions as $c_0 + \sum_i c_i a_i$, where $a_i \in X$ are the program variables, and c_i are unknown integer coefficients we introduce, including a constant term c_0. Note that this is the most general form of any expression possible under linear arithmetic. If some program variable does not appear in the expression, then the solver will instantiate its coefficient to 0. Thus the task of finding an arbitrary expression reduces to finding particular integer constants. A similar approach can encode predicates. We encode each

program predicate, and atomic facts that appear in invariants as $(c_0 + \sum_i c_i a_i) \geq 0$. Again, notice that it is the most general relation within linear arithmetic.

Removing $\forall X$ Converting the formula into one that is purely existential requires a little bit more work. We take a detour into linear algebra, by converting our boolean implications into linear equivalences. Farkas' lemma [52] is a well-known result from linear algebra which states that a boolean implication

$$(\bigwedge_i (e_i \geq 0)) \Rightarrow e \geq 0 \tag{1}$$

holds iff another equivalence

$$(\lambda_0 + \sum_i \lambda_i e_i) \equiv e$$

holds. That is, Farkas' lemma states that a set of linear inequalities with expressions e_i imply another inequality $e \geq 0$ iff e can be expressed as a linear combination of the e_i, modulo some constant. Intuitively, it states that the intersection of a set of halfspaces h_i is contained within another halfspace h iff some halfspace made out a combination of some of h_i (possibly displaced) is identical to h. This elementary result allows us to simplify our program implications. In particular we note the program implications we saw in Section 3 are exactly in the form of Eq. 1; with a little linear algebra to convert each strict inequality into an inequality, and splitting equalities into two inequalities. For example, if we had a program implication $i > 0 \wedge j \geq 0 \Rightarrow \tau$ from a program with variables i and j, and invariant τ that we expanded as above into:

$$i > 0 \wedge j \geq 0 \Rightarrow (c_0 + c_1 i + c_2 j) \geq 0$$

Then we can apply Farkas' lemma to get the equivalence:

$$\lambda_0 + \lambda_1 (i - 1) + \lambda_2 j \equiv (c_0 + c_1 i + c_2 j)$$

One way to remember this translation is to rewrite the equation with $+$ instead of $\wedge$, and $\equiv$ instead of $\Rightarrow$, remove all the ≥ 0, and multiply all left hand side terms with λ_i's and add a constant λ_0.

Note that the λ_i's introduced in the application of Farkas' lemma are real valued (yet-to-be-discovered) constants. Later on we will make an additional assumption of solving for only integer λ_i's, with an additional λ_{rhs} on the right hand side of the equation. This formulation allows for any rational λ_i's, but does not allow for irrationals. We have not found a single instance in practice where this would hurt completeness. Note that for rational λ_i's this formulation is complete.

The application of Farkas' lemma, i.e., a detour into linear algebra, gains us a very significant benefit. The equivalence needs to hold under forall program variables, i.e., $\forall X$. So we can simply collect the coefficients of all the terms, i.e., the program variables, and construct another system of equations with the same solution, except having no universally quantified variables. For our example, we collect the coefficients of the constant term, i, and j to get the following three equations:

$$\begin{aligned} \lambda_0 - \lambda_1 &= c_0 && \ldots \text{for the constant} \\ \lambda_1 &= c_1 && \ldots \text{for } i \\ \lambda_2 &= c_2 && \ldots \text{for } j \end{aligned}$$

This is an equation set from one program implication. There will be similar ones from all other program implications. Now we can go back to our boolean domain, as these are conjunctions of formulae that need to hold. We therefore get a system of equation that is now only existentially quantified over the constants we introduced. In our example, we have a set of equations that are existentially quantified $\exists \lambda_0, \lambda_1, \lambda_2, c_0, c_1, c_2$. That is, we have a satisfiability problem.

We send the satisfiability problem, thus constructed from all program implications, and ask an SMT solver for the solution. The SMT solver gives us a solution for all λ_i and all c_i. The c_i are used to trivially reconstruct the required invariants, while we disregard the λ_i. A more detailed presentation of this reduction and its applications are in previous work [29] and in the first author's Ph.D. thesis [56].

4.1.2 Predicate Abstraction

Predicate abstraction is a technique in which programs are analyzed in terms of a suitably chosen set of predicates. These predicates can be arbitrary relations over program variables, and are potentially significantly more expressive than just linear relations. For instance, it is possible to reason about arrays using `sel` and `upd` predicates, or bit-vector manipulations such as XOR, AND, etc. The downside is that there is more programmer involvement in the process. It is not possible to build the most general template over arbitrary predicates, as we did in linear arithmetic, but instead we have to explicitly enumerate a set of predicates to work with. In our system, we expect the user to specify this set.

Examples of Predicate Sets Earlier in Section 2.1, for `BinarySearch`, we showed the template but not the predicates whose boolean combinations are used to instantiate the holes of the template. The set of predicates is provided by the user as an enumeration of a set of possibilities:

```
enumerate(..) {
1  set1 = {v op v | v ∈ {j, low, high, 0, n}, op ∈ {≤, <}};
2  set2 = {v op v | v ∈ {A[j], A[j − 1], A[j + 1], e, 0},
3                                          op ∈ {≤, ≠}};
4  return set1 ∪ set2;
}
```

Also, in Section 3 we promised to show the atomic relations that populate the invariant template holes for `InsertionSort`. These are enumerated similar to the

above:

```
enumerate(..) {
1  set1 = {v − c op v | v ∈ {x, y, i, j, n},  c ∈ {0, ±1},
2                                              op ∈ {≤, ≥, ≠}};
3  set2 = {v = v | v ∈ {A[t], Ã[t], val | t ∈ {x, y, i, j, n−1}}};
4  return set1 ∪ set2;
}
```

While these predicate sets help narrow the space of possibilities, we still need to tackle the two problems when converting our formulae into satisfiability formulae. We discuss these for the case of predicate abstraction now.

Encoding expressions and predicates For predicate abstraction, the set of expressions and predicates is obtained directly from the programmer. The programmer makes an educated guess about the possible set of expressions and predicates that might be needed in the invariants, and program. This set may be vastly larger than the exact predicates required. The tool will reason about the program and pick the exact predicates to instantiate appropriately. Additionally, this enumeration is global for the entire program's holes, instead of for individual holes. This reduces the programmer's effort by ensuring that they do not have to look into the program or its template to write the predicate enumeration.

Removing $\forall X$ We take a different approach towards taking care of the nested quantification in our program implications. We notice that satisfiability solvers can check tautologies, using a simple trick. If we wish to just infer if $\forall X : f'$ holds, i.e., without any nested quantification, then we can simply ask the solver to find a satisfying assignment to $\exists X : \neg f'$, and if one exists then the tautology does not hold. On the other hand, if the negation is unsatisfiable then the tautology holds.

We cannot apply this trick directly because of the nested quantification in $\exists$`[-]`$\forall X : F$. But for predicate abstraction, we have a set of candidates that can populate the holes. Therefore, we pick a polynomial set of instantiations, and fill out the holes in the program implications in F. Once instantiated, we are left with an implication that is only universally quantified over the program variables. We can check whether that is a tautology using the SMT solver. Making a polynomial number of such instantiations (the formal details of which are in previous work [57] and Ph.D. thesis [56]), we get enough information about individual predicates to set up another satisfiability query to evaluate the real $\exists$`[-]` question.

The way we accumulate the information about individual predicates is in terms of *boolean indicator* variables. Thus, for each predicate p and each hole x in the template, there is a boolean variable $b_{p,x}$. The meaning of this predicate is that if $b_{p,x}$ is *true* then predicate p goes into hole x, else not. Using the polynomial number of SMT queries over the individual program implications we make, we accumulate boolean constraints over these indicator variables. The intuitive idea is that each program implication constrains the booleans in specific ways. The correctness of the program relies on satisfying all program implications simultaneously. This corresponds to conjoining all boolean constraints accumulated for all boolean indicators and finding a single satisfying solution. The satisfying solution assigns truth values to the indicators, which in turn tell us what predicate populates what hole—yielding the invariants. Again, the formal details are in previous work [57] and Ph.D. thesis [56].

4.2 Applying the Solving Technology to Synthesis

The solving technology we had discussed up until now does not make a distinction between invariant and program holes. Especially, in the manner we encode basic blocks as transition systems, statements appear only as equality predicates, and statements with holes are equality predicates with one side a hole, e.g., $j' =$ `[-]`. Thus the approach presented for linear arithmetic and predicate abstraction should suffice to infer programs too. While that is almost correct, it lacks in specific ways. In particular, if we are inferring programs, they have to be of a specific form to be translatable back to imperative code.

4.2.1 Safety + Ranking + Well-formedness

We have up until now encoded safety conditions for the program using correctness assertions. These suffice when we are verifying programs, as termination is ensured externally by the programmer, and if non-terminating then the program satisfies any postcondition trivially. But when generating programs, there are a multitude of solutions that are non-terminating (and so satisfy safety trivially), and so if we ask the solver for a random solution, we are left with many uninteresting programs.

To ensure that we get interesting, terminating programs, we constrain that a valid *ranking function* exist, for each loop. Notice that since we are already inferring expressions elsewhere in the program, this additional expression is inferred in the same framework. For each ranking function, we have to construct constraints similar to safety constraints, but just for the loop body that ensure that in each iteration the ranking function decreases.

Asking for a solution to safety+ranking yields results, but not all these can be translated back into imperative programs. The problem occurs when synthesizing conditionals. For instance, there is template fragment:

$$\texttt{[-]} \rightarrow \texttt{[-]}_X$$
$$\texttt{[-]} \rightarrow \texttt{[-]}_X$$

that is intended to represent a conditional that we hope to translate to a structured `if`$(g)\{s_1\}$`then`$\{s_2\}$. The conditions required for this translation are that the guard

holes should be a tautology, because in the imperative version the guards are g and $\neg g$, respectively for s_1 and s_2, and the guards together form a tautology. Additionally the statements inferred should not be $false$, because a input-output transition system (relation between primed and unprimed variables) can never be $false$. We call these two constraints *well-formedness* constraints. The formal description of these constraints are available in previous work [59].

Example Putting all of safety, ranking, and well-formedness constraints together and solving using the satisfiability-based techniques we have described does yield, well-formed terminating programs. For example, for `BresenhamsTemplate` in Section 2.1.2 the solver generates the following intermediate program after solving using linear arithmetic:

```
BresenhamsIntermediate(int X, Y) {
  Assert 0 < Y ≤ X
  true → v1' = 2Y − X ∧ y' = 0 ∧ x' = 0
  while (x ≤ X) {
      v1 < 0 → v1' = v1 + 2Y ∧ x' = x + 1
      v1 ≥ 0 → v1' = v1 + 2(Y−X) ∧ y' = y + 1 ∧ x' = x + 1
  }
}
```

The solver additionally generates the invariants and ranking function shown earlier. For these inferred transition systems and loop guards, a little bit of postprocessing yields the program `Bresenhams` shown earlier.

Additional constraints Notice that we have incrementally constrained the space of programs with each addition of correctness, well-formedness, and termination constraints. While these are arguably the smallest set of constraints that are guaranteed to give us useful programs, there are other constraint sets that might be of interest, e.g., performance constraint, memory usage constraints etc. These can be imposed in addition, if we were to for the moment ignore the hardness of formalizing them. While the search space might be narrower with additional constraints, this would not necessarily lead to faster solving. In our experience, we have found that while at the start additional constraints help the solver narrow into a solution faster, excessive constraints lead to excessive backtracking, thus higher solving times. Due to these counteracting factors, our system currently imposes the minimal set of correctness, well-formedness, and termination constraints. We leave the decision to either filter candidates post-generation, or through additional constraints for future work, as may be required for performance, or memory considerations, for instance.

Compositionality: Synthesis vs Verification A point similar to the above is that of compositionality of synthesis. In verification using traditional iterative approaches composition of proofs is linear time. That is, proving two properties takes only as much effort as the sum of proving each individually. On the other hand, in most synthesis approaches including ours, such compositionality properties do not exist. While this is true, it is also the case that our satisfiability/template-based verification does not possess the linear time composition properties of traditional verification approaches. Thus we trade-off some properties in the easier problem of verification, such that synthesis is a natural extension. Compositional/modular program synthesis is a very exciting open research question.

5 Case Study: Verifying and Synthesizing Selection Sort

We now present a case study on our experience with using template-based verification and synthesis tools.

5.1 Verification

Let us first consider analyzing selection sort:

```
   SelectionSort(Array A, int n) {
1    i := 0;
2    while (i < n − 1) {
3       min := i; j := i + 1;
4       while (j < n) {
5          if (A[j] < A[min]) min := j;
6          j := j + 1;
7       }
8       Assert(i ≠ min);
9       if (i ≠ min) swap(A, i, min);
10      i := i + 1;
11   }
   }
```

where the function `swap` is syntactic sugar in the template language that gets expanded to a swapping operation over integers using a temporary.

Suppose we wish to maximize the number of times the sorting algorithm swaps elements, i.e., exhibits its worst case number of swaps. In particular, we want to infer an input over which the algorithm swaps every time it can on line 9. We can use our verification tools to generate the maximally-weak precondition corresponding to that, by using the assert on line 8. We know that to relate any global precondition and some internal assertion, we would need appropriate loop invariants to summarize the behavior of the loops. We therefore pick a template that allows summarizing properties of sets of array elements:

```
   template(..) {
1    fact1 = [-];
2    fact2 = true; foreach ({k | [-]})
3                     { fact2 = fact2 ∧ [-]}
4    fact3 = true; foreach ({k | [-]})
5                     { fact3 = fact3 ∧ [-]}
6    fact4 = true; foreach ({k1, k2 | [-]})
7                     { fact4 = fact4 ∧ [-]}
8    return fact1 ∧ fact2 ∧ fact3 ∧ fact4;
   }
```

This template encodes four conjuncts

$$\begin{array}{l}\texttt{[-]} \wedge \\ (\forall k : \texttt{[-]} \Rightarrow \texttt{[-]}) \wedge \\ (\forall k : \texttt{[-]} \Rightarrow \texttt{[-]}) \wedge \\ (\forall k_1, k_2 : \texttt{[-]} \Rightarrow \texttt{[-]})\end{array}$$

Since we wish to infer array properties, we choose to use predicate abstraction to infer our precondition. For predicate abstraction, we additionally need to specify a set of predicates. We enumerate two sets: `set1` represents linear inequalities between program variables (with an optional constant c difference), and `set2` represents linear inequalities between array values:

```
enumerate(..) {
1  set1 = {v − c  op  v | v ∈ {0, k, k1, k2, i, j, min, n},
2                          c ∈ {0, 1}, op ∈ {≤, ≥, >}};
3  set2 = {A[v]  op  A[v] | v ∈ {0, k, k1, k2, i, j, min, n−1},
4                                          op ∈ {≤, <}};
5  return set1 ∪ set2;
}
```

Notice that in `set2` we use $n-1$ as n cannot be a valid index into the array. Using this set we run our template-based verifier and it infers the following loop invariants. For the outer loop it infers:

$$\begin{pmatrix}\forall k_1, k_2 : (i \leq k_1 < k_2 < n-1) \Rightarrow A[k_1] < A[k_2] \\ \forall k : i \leq k < n-1 \Rightarrow A[n-1] < A[k]\end{pmatrix}$$

For the inner loop it infers:

$$\begin{pmatrix}\forall k_1, k_2 : (i \leq k_1 < k_2 < n-1) \Rightarrow A[k_1] < A[k_2] \\ \forall k : (i \leq k < n-1) \Rightarrow A[n-1] < A[k] \\ \forall k : (i \leq k < j) \Rightarrow A[\texttt{min}] \leq A[k] \\ j > i \wedge i < n-1\end{pmatrix}$$

But more importantly, the tool infers the following non-trivial maximally-weak precondition:

$$\begin{pmatrix}\forall k : (0 \leq k < n-1) \Rightarrow A[n-1] < A[k] \\ \forall k_1, k_2 : (0 \leq k_1 < k_2 < n-1) \Rightarrow A[k_1] < A[k_2]\end{pmatrix}$$

This states the somewhat surprising precondition that selection sort exhibits its worst case number of swaps when the input array is essentially strictly increasingly sorted (second quantified fact), except that the last element is strictly smaller than the rest (first quantified fact). Automation and formal verification guarantees are required when programs get complicated and we need inference for such involved preconditions.

5.2 Synthesis

While verification is informative, we wondered if our synthesis system built on top of verification, could in fact generate the sorting algorithm from high level hints provided by the programmer, in this case us. We first needed to come up with a program template. We decided on a few things to start with. First, we wanted a loop-based, as opposed to recursive sort. Second, elementary knowledge about algorithms told us that we would need at least two loops, with loop counters, i_1 and i_2, respectively. Third, we did not want to get into problems with the program losing or replicating elements, and so chose the primitive operation to be a swap. Therefore, we constructed the following program template:

```
SelSort(Array A, int n) {
1  [-]_{i1}
2  while (i1 <[-]) {
3     [-]_{i2}
4     while (i2 <[-]) {
5        [-]→ swap(A, [-], [-]); [-]_{v1}
6        i2++;
7        }
8     i1++;
9  }
10 Assert sorted(A, n); return A;
}
```

Notice that on line 5 we have introduced another variable v_1. Synthesis attempts without it failed. At that point the options available to user of the tool are to either modify the control flow template or add auxiliary variables. We conjectured that an auxiliary variable would be required and updated in the portion of the program inside the loop where the real work was being performed.

Of course, we need the sortedness specification encoding $\forall k_1, k_2 : (0 \leq k_1 < k_2 < n) \Rightarrow A[k_1] \leq A[k_2]$ as:

```
sorted(Array A, int n)
1  fact = true;
2  for(k1 = 0; k1 < n; k1++)
3     for(k2 = k1 + 1; k2 < n; k2++)
4        fact = fact ∧ A[k1] ≤ A[k2];
5  return fact;
}
```

We ran these templates through the tool, and it produced selection sort, along with its invariants and ranking functions. But it required some manual tweaks. The initial attempt failed, subsequent to which we realized that the program may require a loop that is slightly non-uniform, a case which we had experienced before. In some programs the loops are not uniform, and in particular the first and last iteration of the loop tend to be different. Therefore at times synthesis requires replicating the body of the loop in the template. In the case of our template, the body "$\texttt{[-]} \rightarrow \texttt{swap}(A, \texttt{[-]}, \texttt{[-]}); \texttt{[-]}_{v_1}$" is replicated before and after the loop. Thus the tool generates the following intermediate representation (still a template but elaborated):

```
SelSort(Array A, int n) {
1   [-]_{i1}
2   while (i1 <[-]) {
3      [-]_{i2}
4      [-]→ swap(A, [-], [-]); [-]_{v1}
5      while (i2 <[-]) {
6         [-]→ swap(A, [-], [-]); [-]_{v1}
7         i2++;
8         }
9      [-]→ swap(A, [-], [-]); [-]_{v1}
10     i1++;
11  }
12  Assert sorted(A,n); return A;
}
```

It then proceeds to solve for the holes, using the predicate sets as earlier and some extra predicates for expressions, to get the following intermediate solution:

```
SelSort(Array A, int n) {
1   i1 := 0
2   while (i1 < n − 1) {
3      i2 := i1 + 1
4      true→ swap(A,0,0); v1 := i1
5      while (i2 < n) {
6         (A[i2] < A[v1]) → swap(A,0,0); v1 := i2
7         i2++;
8         }
9      true→ swap(A,i1,v1); v1 := v1
10     i1++;
11  }
12  Assert sorted(A,n); return A;
}
```

Manually cleaning up the redundant swaps and redundant assignments, we get the following program:

```
SelSort(Array A, int n) {
1   i1:=0;
2   while^{τ1,φ1}(i1 < n − 1) {
3      i2:=i1+1;
4      v1:=i1;
5      while^{τ2,φ2}(i2 < n) {
6         if (A[i2]<A[v1]) v1:=i2;
7         i2++;
8      }
9      swap(A,i1,v1);
10     i1++;
11  }
12  return A;
}
```

The tool also outputs the ranking function φ_1 and loop invariant τ_1 for the outer loop (with counter i_1):

φ_1: $n - i_1 - 2$

τ_1: $\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \wedge\ k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]$

and the ranking function φ_2 and loop invariant τ_2 for the inner loop (with counter i_2):

φ_2: $n - i_2 - 1$

$$\tau_2: \begin{pmatrix} i_1 < i_2 \wedge i_1 \leq v_1 < n \\ \forall k_1, k_2 : 0 \leq k_1 < k_2 < n \wedge\ k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2] \\ \forall k : i_1 \leq k < i_2 \wedge k \geq 0 \Rightarrow A[v_1] \leq A[k] \end{pmatrix}$$

5.3 Discussion: Benefits and Limitations of a Template-based/Satisfiability-based approach

It is important to compare the benefits and limitations of our approach against traditional iterative fixed-point approximation techniques, such as data-flow analyses and abstract interpretation.

The key difference between a satisfiability-based approach and traditional techniques is the lack of iterative approximations. By encoding the problem as a solution to a SAT instance, we are able to delegate fixed-point solving to the SAT solver, and verification is non-iterative, and so correspondingly is synthesis.

We note two advantages of a satisfiability-based approach. First, a satisfiability-based approach is goal-directed and hence has the potential to be more efficient. The data-flow analyses or abstract interpreters typically work either in a forward direction or in a backward direction, and hence are not goal-directed. Some efforts to incorporate goal-directedness involve repeatedly performing a forward (or backward) analysis over refined abstractions obtained using counterexample guidance, or by repeatedly iterating between forward and backward analyses [10]. However, each forward or backward analysis attempts to compute the most precise information over the underlying domain, disregarding what might really be needed. On the other hand, the satisfiability-based approach is goal-directed; it abstracts away the control-flow of the program and incorporates information from both the precondition as well as the postcondition in the constraints. Second, a satisfiability-based approach does not require widening heuristics, that can lead to uncontrolled loss of precision, but are required for termination of iterative fixed-point techniques. Abstract interpreters iteratively compute approximations to fixed-points and use domain-specific extrapolation operators (widening) when operating over infinite height lattices (e.g., lattice of linear inequalities) to ensure termination. Use of widening leads to an uncontrolled loss of precision. This has led to development of several widening heuristics that are tailored to specific classes of programs [63, 23, 20, 21]. Our technique on the other hand, can handle all the programs they can handle by just using appropriate templates, and without specialization of the theory or implementation of the analysis.

We now note some limitations of a satisfiability-based approach. First, the execution time of analyses in this framework is expected to be less deterministic as it depends on the underlying SAT solver. As yet, we have not encountered such brittleness for our benchmarks, but anomalous runtimes may appear in pathological examples. Second, a domain-specific approach, e.g., Farkas' lemma for linear arithmetic, or our novel reduction for predicate abstraction, is needed to reduce constraints to SAT. We require such domain specific reductions as do previous techniques (e.g., join, widen, and transfer functions in abstract interpretation). Domain specifications

help make the problem tractable from what is otherwise an undecidable problem. The key to successfully exploiting the power of a satisfiability-based framework for program analysis will be the development such domain-specific reductions. Lastly, we note that comparisons with other techniques are necessarily orthogonal as our technique relies on user-provided templates while previous techniques attempt to be fully-automated. We perceive this to be an advantage since this allows the user to choose the level of expressivity they needs for analyzing/synthesizing their program, without having to modify the development tools themselves.

6 Evaluation

We built our tools as part of the VS^3 (Verification and Synthesis using SMT Solvers) project that implement the techniques described here. We ran our experiments on a 2.5GHz Intel Core 2 Duo machine with 4GB of memory. We use Z3 version 1.2 SMT solver.

6.1 Verification

We consider small but complicated programs that manipulate unbounded data structures. These programs have been considered in state-of-the-art alternative techniques that infer data-sensitive properties of programs.

In each verification instance, the programmer supplies the program as C code, and additionally templates that guide the verification process. For verification, our tool requires templates for invariants, which are typically composed of the quantifier structure and usually nested within a "$[-] \Rightarrow [-]$" term. Additionally, to populate the holes the programmer supplies an enumeration of predicates derived by separate program variables and trivial constants using relational operators. These typically total about 10 lines of extra annotations for each instance.

Simple array/list manipulation Table 1 presents the assertions that we prove for some small benchmarks that are difficult for mechanical verification, along with runtimes using our tool. By adding axiomatic support for reachability $\rightsquigarrow$, we were able to verify simple list programs.

Consumer Producer [37] is a program loop that nondeterministically writes (produces) a new value into a buffer at the head or reads (consumes) a value at the tail; we verify that the values read by the consumer are exactly those that are written by the producer. Partition Array [3,37] splits an array into two separate arrays, one containing the zero entries and the other the non-zero; we verify that the resulting arrays indeed contain zero and non-zero entries. List Init [28] initializes the *val* fields of a list to 0; we verify that every node reachable from the head has been initialized. List Delete [28] (respectively, List Insert [28]) assumes a properly initialized list and deletes (respectively inserts) a properly initialized node; we verify that the resulting lists still have *val* fields as 0.

Sortedness property Sorting benchmarks are some of the hardest verification instances for array programs. We verify sortedness for all major sorting procedures. Table 2 presents the assertions that we proved for these procedures, and the corresponding runtimes.

We evaluate over selection sort, insertion sort and bubble sort (one that iterates n^2 times irrespective of array contents, and one that maintains a flag indicating whether the inner loop swapped any element or not, and breaks if it did not). For quick sort and merge sort we consider their partitioning and merge steps, respectively.

The reader may be wondering why the semantically equivalent, but syntactically different postconditions are used in the different versions of sorting. The answer lies in the way the proof of correctness, i.e., the inductive invariants, for each procedure are structured. One set requires a simple induction hypothesis, while the other requires complete/strong induction. Since the SMT-solver is not capable of generalizing from a single step to multiple steps, we need to help it out by giving it either a doubly quantified template, or a singly quantified template.

$\forall\exists$ *properties:* Our tools additionally can be used to prove that under the assumption that the elements of the input array are distinct, the sorting algorithms do not lose any elements of the input. Table 3 shows the $\forall\exists$ facts it proves along with the runtimes required to infer the invariants corresponding to them.

Worst-case upper bounds: Not only can our tool prove sortedness and input preservation, it can also derive inputs (as preconditions) that lead to worst-case execution times for the given algorithm. We generate the maximally weak preconditions, i.e., worst-case inputs, for each of the sorting examples as shown in Table 4 along with their runtimes. Notice that the inner loop of merge sort and the n^2 version of bubble sort always perform the same number of writes, and therefore no assertions are present and the precondition is *true*.

Functional correctness: Often, procedures expect conditions to hold on the input for functional correctness. These can be met by initialization, or by just assuming facts at entry. We consider the synthesis of the maximally weak such conditions. Table 5 lists our programs, the interesting non-trivial preconditions (*pre*) we compute under the functional specification (*post*) supplied as postconditions, along with their runtimes. (We omit other non-interesting preconditions that do not give us more insights into the program but are generated by the

Benchmark	Assertion proved	Time (s)
Consumer Producer	$\forall k : 0 \leq k < n \Rightarrow C[k] = P[k]$	0.45
Partition Array	$\forall k : 0 \leq k < j \Rightarrow B[k] \neq 0$ $\forall k : 0 \leq k < l \Rightarrow A[k] = 0$	0.15
List Init	$\forall k : x \rightsquigarrow k \wedge k \neq \perp \Rightarrow k \rightarrow val = 0$	0.06
List Delete	- same as above -	0.03
List Insert	- same as above -	0.12

Table 1. The assertions proved for verifying simple array/list programs.

Benchmark	Assertion proved	Time (s)
Selection Sort	$\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2]$	1.32
Bubble Sort (n^2)	- same as above -	0.47
Insertion Sort	$\forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1]$	2.90
Bubble Sort (flag)	- same as above -	0.16
Quick Sort (inner)	$\forall k : 0 \leq k < pivot \Rightarrow A[k] \leq pivotval$ $\forall k : pivot < k < n \Rightarrow A[k] > pivotval$	0.43
Merge Sort (inner)	under sorted input arrays A, B, output C is sorted	2.19

Table 2. The assertions proving that sorting programs output sorted arrays.

Benchmark	Assertion proved	Time (s)
Selection Sort	$\forall y \exists x : 0 \leq y < n \Rightarrow \tilde{A}[y] = A[x] \wedge 0 \leq x < n$	17.02
Insertion Sort	- same as above -	2.62
Bubble (n^2)	- same as above -	1.10
Bubble (flag)	- same as above -	1.56
Quick Sort (inner)	- same as above -	1.83
Merge Sort (inner)	$\forall y \exists x : 0 \leq y < m \Rightarrow A[y] = C[x] \wedge 0 \leq x < t$ $\forall y \exists x : 0 \leq y < n \Rightarrow B[y] = C[x] \wedge 0 \leq x < t$	7.00

Table 3. The assertions proved for verifying that sorting programs preserve the elements of the input. $\tilde{A}$ is the array A at the entry to the program.

Benchmark	Precondition inferred	Time (s)
Selection Sort	$\forall k : 0 \leq k < n-1 \Rightarrow A[n-1] < A[k]$ $\forall k_1, k_2 : 0 \leq k_1 < k_2 < n-1 \Rightarrow A[k_1] < A[k_2]$	16.62
Insertion Sort	$\forall k : 0 \leq k < n-1 \Rightarrow A[k] > A[k+1]$	39.59
Bubble Sort (flag)	$\forall k : 0 \leq k < n-1 \Rightarrow A[k] > A[k+1]$	9.04
Quick Sort (inner)	$\forall k_1, k_2 : 0 \leq k_1 < k_2 \leq n \Rightarrow A[k_1] \leq A[k_2]$	1.68

Table 4. The preconditions inferred by our algorithms for worst case upper bounds runs of sorting programs.

Benchmark	Preconditions inferred under given postcondition	Time (s)
Partial Init	*pre:* (a) $m \leq n$ (b) $\forall k : n \leq k < m \Rightarrow A[k] = 0$ *post:* $\forall k : 0 \leq k < m \Rightarrow A[k] = 0$	0.50
Init Synthesis	*pre:* (a) $i = 1 \wedge max = 0$ (b) $i = 0$ *post:* $\forall k : 0 \leq k < n \Rightarrow A[max] \geq A[k]$	0.72
Binary Search	*pre:* $\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \Rightarrow A[k_1] \leq A[k_2]$ *post:* $\forall k : 0 \leq k < n \Rightarrow A[k] \neq e$	13.48
Merge	*pre:* $\forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1]$ $\forall k : 0 \leq k < m \Rightarrow B[k] \leq B[k+1]$ *post:* $\forall k : 0 \leq k < t \Rightarrow C[k] \leq C[k+1]$	3.37

Table 5. Given a functional specification (post), the maximally weak preconditions (pre) inferred by our algorithms for functional correctness.

tool nonetheless while enumerating maximally weak preconditions.)

Array Init initializes the locations $0 \ldots n$ while the functional specification expects initialization from $0 \ldots m$. Our algorithms, interestingly, generate two alternative preconditions, one that makes the specification expect less, while the other expects locations outside the range to be pre-initialized. Init Synthesis computes the index of the maximum array value. Restricting to equality predicates we compute two incomparable preconditions that correspond to the missing initializers, for the loop counter variable i and location max of the maximum element. Notice that the second precondition is indeed maximally weak for the specification, even though max could be initialized out of bounds. If we expected to strictly output an array index and not just the location of the maximum, then the specification should have contained $0 \leq max < n$. Binary Search is the standard binary search for the element e with the correctness specification that if the element was not found in the array, then the array does not contain the element. We generate the precondition that the input array must have been sorted. Merge Sort (inner) outputs a sorted array. We infer that the input arrays must have been sorted for the procedure to be functionally correct.

Benchmark	Verif.	Synthesis
Swap two	0.11	0.12
Strassen's	0.11	4.98
Sqrt (linear search)	0.84	9.96
Sqrt (binary search)	0.63	1.83
Bresenham's	166.54	9658.52
Bubble Sort	1.27	3.19
Insertion Sort	2.49	5.41
Selection Sort	23.77	164.57
Merge Sort	18.86	50.00
Quick Sort	1.74	160.57
Fibonacci	0.37	5.90
Checkerboard	0.39	0.96
Longest Common Subseq.	0.53	14.23
Matrix Chain Multiply	6.85	88.35
Single-Src Shortest Path	46.58	124.01
All-pairs Shortest Path[1]	112.28	(i) 226.71 (ii) 750.11

Table 6. Experimental results for proof-theoretic synthesis for (a) Arithmetic benchmarks (b) Sorting benchmarks, and (c) Dynamic Programming. For each category, we indicate the tool used to solve the verification conditions and the synthesis conditions.

6.2 Synthesis

Table 6 presents the runtimes of our synthesizer over arithmetic, sorting and dynamic programming benchmarks. We measure the time for verification and the time for synthesis using the same tool. The benchmarks that we consider are difficult even for verification. Consequently the low average runtimes for proof-theoretic synthesis are encouraging. Synthesis takes on average only 17.36 times and median 6.68 times longer than verification, which is encouraging and shows that we can indeed exploit the advances in verification to our advantage for synthesis.

For each synthesis instance, the programmer supplies the scaffold, i.e., the program template which is usually very few lines of code, since a single placeholder hole gets instantiated to an entire acyclic fragment of code in the final synthesized output. Additionally they supply predicates and invariant templates as is the case for verification. Again, these extra annotations are typically about 10 lines of code. Statements, which are instantiated as equality predicates are encapsulated within the predicate set as equality relations between program variables and expressions.

Algorithms that use arithmetic We chose a set of arithmetic benchmarks with simple-to-state functional specifications but each containing some tricky insight that human programmers may miss.

- *Swapping without Temporaries* We synthesize a program that swaps two integer-valued variables *without* using a temporary. We give the synthesizer the precondition $(x = c_2 \wedge y = c_1)$ and the postcondition $(x = c_1 \wedge y = c_2)$.
- *Strassen's* 2×2 *Matrix Multiplication* We synthesize a program for Strassen's matrix multiplication, which computes the product of two $n \times n$ matrices in $\Theta(n^{2.81})$ time instead of $\Theta(n^3)$. The key to this algorithm is an acyclic fragment that computes the product of two 2×2 input matrices $\{a_{ij}, b_{ij}\}_{i,j=1,2}$ using 7 multiplications instead of the expected 8. Used recursively, this results in asymptotic savings. The key insight of the algorithm lies in this core. Recursive block multiplication was well known, and Strassen augmented it with an efficient core. We synthesize the crucial acyclic fragment, which is shown in Figure 2. Here the precondition is `true` and the postcondition is the conjunction of four equalities as (over the outputs $\{c_{ij}\}_{i,j=1,2}$):

$$\begin{pmatrix} c_{11}\ c_{12} \\ c_{21}\ c_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21}\ a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21}\ a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

- *Integral Square Root* We synthesize a program that computes the integral square root $\lfloor \sqrt{x} \rfloor$ of a positive number x using only linear or quadratic operations. The precondition is $x \geq 1$ and the postcondition over the output i is $(i-1)^2 \leq x < i^2$. Our synthesizer generates two versions, corresponding to a linear search and a binary search for the integral square root.

[1] These timings are for separately (i) synthesizing the loop guards, and (ii) synthesizing the acyclic fragments. We fail to synthesize the entire program, but with these hints provided by the user, our synthesizer can produce the remaining program.

```
Strassens(int a_ij, b_ij) {
  v1 := (a11+a22)(b11+b22)
  v2 := (a21+a22)b11
  v3 := a11(b12-b22)
  v4 := a22(b21-b11)
  v5 := (a11+a12)b22
  v6 := (a21-a11)(b11+b12)
  v7 := (a12-a22)(b21+b22)
  c11 := v1+v4-v5+v7
  c12 := v3+v5
  c21 := v2+v4
  c22 := v1+v3-v2+v6
  return c_ij;
}
```

Fig. 2. Synthesis result for Strassen's Matrix Multiplication using the arithmetic solver.

```
SelSort(int A[], n) {
  i1 := 0;
  while^{τ1,φ1} (i1 < n − 1)
  |  v1 := i1;
  |  i2 := i1+1;
  |  while^{τ2,φ2} (i2 < n)
  |  |  if (A[i2] < A[v1])
  |  |        v1 := i2;
  |  |  i2++;
  |  swap(A[i1], A[v1]);
  |  i1++;
  return A;
}
```

Ranking functions:
$\varphi_1 : n - i_1 - 2$
$\varphi_2 : n - i_2 - 1$
Invariant τ_1:
$\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \wedge\ k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]$
Invariant τ_1:
$i_1 < i_2 \wedge i_1 \leq v_1 < n$
$\forall k_1, k_2 : 0 \leq k_1 < k_2 < n \wedge\ k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]$
$\forall k : i_1 \leq k < i_2 \wedge k \geq 0 \Rightarrow A[v_1] \leq A[k]$

Fig. 3. Synthesizing Selection Sort. For presentation, we omit degenerate conditional branches, i.e. `true`/`false` guards, We name the loop iteration counters $L = \{i_1, i_2, ..\}$ and the temporary stack variables $T = \{v_1, v_2, ..\}$.

– *Bresenham's Line Drawing Algorithm* We discussed Bresenham's line drawing algorithm earlier. Our synthesizer cannot synthesize the algorithm with the full set of holes as described earlier. We therefore guessed some of the values for the easiest of the unknown holes, e.g., that $x := 0; y := 0; v_1 := 2Y - X$ at initialization and that $x++$ is present within the loop, and one of the guards is $e \geq 0$ and $v_1 := v_1 + 2Y$, and then the synthesizer generates the rest. Given the difficulty of this benchmark, we use it to explore the limitations of our synthesizer (and current SMT solvers). We submitted versions, that successively contain more unknowns, as hard benchmarks to the SMT solving community and they have been incorporated in the latest SMTCOMP competitions.

Sorting Algorithms The sortedness specification consists of the precondition `true` and the postcondition $\forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k+1]$. The full functional specification would also ensure that the output array is a permutation of the input, but verifying—and thus, synthesizing—the full specification is outside the capabilities of most automated tools today. By only allowing swap and move operations we restrict the space of possible programs and can thus use only the sortedness condition as the specification.

Our synthesizer generates the algorithms for non-recursive sorting algorithms and also the main bodies of the recursive sorting algorithms.

Dynamic Programming Algorithms We choose all the textbook dynamic programming examples [8] and attempt to synthesize them from their functional specifications.

The first hurdle (even for verification) for these algorithms is that the meaning of the computation is not easily specified. We need a mechanism that makes the solver aware of the inductive definitions declaratively. We do this by encoding the inductive definitions as recursively specified axioms over symbols which encode the function. To address this issue, we need support for axioms, which are typically recursive definitions.

Our tool allows the user to define the meaning of a computation as an uninterpreted symbol, with (recursive) quantified facts defining the semantics of the symbol axiomatically. For example, the semantics of Fibonacci are defined in terms of the symbol `Fib` and the axioms:

$$\texttt{Fib}(0) = 0$$
$$\texttt{Fib}(1) = 1$$
$$\forall k : k \geq 0 \Rightarrow \texttt{Fib}(k+2) = \texttt{Fib}(k+1) + \texttt{Fib}(k)$$

The tool passes the given symbol and its definitional axioms to the underlying theorem prover (Z3 [13]), which assumes the axioms before every theorem proving query. This allows the tool to verify dynamic programming programs.

– *Fibonacci* We synthesize a program for computing the nth Fibonacci number. Our synthesizer generates a program that memoizes the solutions to the two subproblems $\texttt{Fib}(i_1)$ and $\texttt{Fib}(i_1 + 1)$ in the i_1th iteration. The solver does not have the opportunity to synthesize a recursive more naive program because the template enforces an iterative solution. Figure 4 shows the program.
– *Checkerboard* Our synthesizer generates a program for computing the least-cost path in a rectangular grid (with costs at each grid location), from the bottom row to the top row.

```
Fib(int n) {
  v1:=0; v1:=1; i1:=0;
  while^{τ,φ}(i1 ≤ n)
    | v1:=v1+v2; swap(v1, v2);
    | i1++;
  return v1;
}
```

Ranking function φ:
$x - s$
Invariant τ:
$v_1 = \texttt{Fib}(i_1) \wedge v_2 = \texttt{Fib}(i_1{+}1)$

Fig. 4. Synthesis results for a dynamic programming program, Fibonacci. Here, we name the loop iteration counters $L = \{i_1, i_2, ..\}$ and the temporary stack variables $T = \{v_1, v_2, ..\}$.

- *Longest Common Subsequence (LCS)* Our synthesizer generates a program for computing the longest common substring that appears in the same order in two given input strings (as arrays of characters).
- *Matrix Chain Multiply* Our synthesizer generates a program for computing the optimal way to multiply a matrix chain. Depending on the bracketing, the total number of multiplications varies. We wish to find the bracketing that minimizes the number of multiplications.
- *Single Source Shortest Path* Our synthesizer generates a program for computing the least-cost path from a designated source to all other nodes where the weight of edges is given as a cost function for each source and destination pair.
- *All-pairs Shortest Path* Our synthesizer generates a program for computing all-pairs shortest paths using a recursive functional specification similar to the one we used for single source shortest path. Our synthesizer times out for this example. We therefore attempt synthesis by (i) specifying the acyclic fragments and synthesizing the guards, and (ii) specifying the guards and synthesizing the acyclic fragments. In each case, our synthesizer generates the other component, corresponding to Floyd-Warshall's algorithm.

6.3 Limitations and Scalability

To explore the scalability and limitations of our synthesis approach, we took versions of Bresenham's and Strassen's bechmarks by successively making the synthesis task harder. For Bresenham's benchmark, we start with a template with a single unknown and successively add unknowns noting the synthesis times for each until the tool fails. For Strassen's our synthesizer easily solves the case that Strassen proposed for multiplication with 7 intermediate values. We attempt to synthesize with 6 intermediate values by block splitting using a 3x3 grid (as opposed to 2x2) and asking for a solution using 16 intermediates. If such a solution existed it would yield matrix multiplication in better than $O(n^{2.51})$ complexity (which is the theoretically best known to date using the Coppersmith-Winograd algorithm).

Table 7 shows the result of this experiment. We note that our synthesizer times out for Bresenham's with the full set of unknowns. For the four initial versions each additional unknown increases the synthesis time. For Strassen's our synthesizer reinvents the known algorithm but fails to find ones with better complexity bounds. Lastly, since the solving time is entirely dependent on the efficiency of the solver, we submitted these benchmarks to the SMT competition (SMTCOMP) and all except the last benchmark were incorporated in the BitVector category in the 2010 competition.

7 Related Work

The problem of verification has been well studied, and so we restrict our comparisons to other techniques which have either similar inputs, expressivity goals or methodology. Program synthesis is not as widely studied, and we discuss our approach in the context of the spectrum defined by inductive and deductive approaches at either end. For a more detailed survey, see [24].

7.1 Verification

Approaches similar in input: Templates for invariant inference The use of templates for restricting the space of invariants is not entirely new [5,50]; although defining them as explicitly as we do here, and leveraging satisfiability-solving, is new. In fact, *domains* in abstract interpretation [9] are templates of sorts. Abstraction refinement techniques have also used template to instantiate proof terms [34]. Lately, *refinement templates* have been used for inferring limited dependent types [49]. There have been various approaches in the past incorporating placeholder/holes into languages with various different objectives, but the common theme of facilitating the programmer's task in writing specifications [48], program refinements [14], axiomatic definition [38], transformational programming [1].

Approaches with similar expressivity goals: Quantified invariants Quantification in invariants is critical for verifying important properties of programs. In fact, sorting programs are the staple benchmarks for the verification community precisely because they require complicated quantified invariants. Previous proposals for handling quantification attempted to either specialize the analysis depending on the quantification present resulting in complicated decision procedures [28], or the full literal specification of quantified predicates [11], or use implicit

Benchmark	Synthesis Time	Unknowns	SMTCOMP
Bresenham's v1	11 mins	1 expr	'10
Bresenham's v2	45 mins	2 expr	'10
Bresenham's v3	114 mins	3 expr	'10
Bresenham's v4	160 mins	4 expr	'10
Bresenham's v5	Timeout	5 expr	'10
Bresenham's v6	Timeout	4 expr + 1 guard	'09/'10
Bresenham's v7	Timeout	6 expr + 1 guard	'10
Strassen's v1	13 secs	with 7 intermediate	'09/'10
Strassen's v2	Timeout	with 6 intermediate	'10
Strassen's v3	Memout	16 intermediates (3x3 grid)	-

Table 7. Gradations of the Bresenham's and Strassen's benchmarks. Solving times reported are for Z3 version 1.2. We additionally submitted the benchmarks to the SMT competition SMTCOMP. All but one were included in the 2010 SMTCOMP (http://www.smtexec.org/exec/?jobs=684). Two were included in the 2009 SMTCOMP (http://www.smtexec.org/exec/?jobs=529). In 2009 the winner was able to solve S1 but not A12. In 2010 the winner was additionally able to solve A7.

quantification through free variables for limited properties [40,42,41,18]. Our approach is arguably more robust because we delegate the concern of reasoning about quantification to SMT solvers, which have been well engineered to handle quantified queries that arise in practice [12]. As the handling of quantification gets more robust in these solvers, our tools will benefit. Even with the current technology, quantification handling was robust for even the most difficult verification examples.

Approaches similar in methodology: Satisfiability-based invariant inference Satisfiability-based invariant inference for linear arithmetic has been explored by others in developing efficient program analyses. InvGen generates SAT instances that are simpler to solve by augmenting the core constraints with constraints over a set of symbolic paths derived from test cases [31,32]. Satisfiability-based solutions find applications in hardware synthesis as well [7]. The satisfiability-based approach we propose was inspired by constraint-based analyses proposed earlier [5,50,51]. Additionally, by unrolling loops, and bit-blasting data structures, a previous approach encodes the existence of a bug as SAT [64], but it differs from our approach in not addressing correctness.

7.2 Synthesis

Deductive Synthesis Deductive synthesis is an approach to synthesis that generates programs through iterative refinement of the specification. At each step of the refinement, well-defined proof rules are used, each of which corresponds to the introduction of a programming construct. For instance, case splits in the proof leads to a conditionals in the program, induction in the program leads to loops in the program. Deductive synthesis was explored in work by Manna, Waldinger and others in the 1960's and 1970's [44]. The core idea was to *extract* the program from the *proof* of realizability of the formula $\forall \mathbf{x} : \exists \mathbf{y} : pre(\mathbf{x}) \Rightarrow post(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ and $\mathbf{y}$ are the input and output vectors, respectively [22,62]. The problem is that these approaches provide little automation and additionally involve the programmer in the proof-refinement, which experience has shown is hard to achieve.

Most of the work in deductive synthesis stems from the seminal work of Manna and Waldinger [43,44]. Successful systems developed based on this approach include Bates and Constable's NuPRL [6] system, and Smith's KIDS [53], Specware [47], and Designware [54]. In these systems, the synthesizer is seen as a compiler from a high-level (possibly non-algorithmic) language to an executable (algorithmic) language, guided by the human. To quote Smith, "the whole history of computer science has been toward increasingly high-level languages--machine language, assembler, macros, Fortran, Java and so on—and we are working at the extreme end of that."

While such systems have been successfully applied in practice, they require significant human effort, which is only justified for the case of safety/mission-critical software [16]. As such, these systems can be viewed as programming aids for these difficult software development tasks, somewhat related to the idea of domain-specific synthesizers such as AutoBayes for data-analysis problems [17], StreamIt for signal-processing kernels [60], or Simulink for synthesis of control systems and digital signal processing applications [2].

Our approach can be seen as midway between deductive and schema-guided synthesis. Schema-guided synthesis takes a template of the desired computations and generates a program using a deductive approach [19]. Some heuristic techniques for automating schema-guided synthesis have been proposed, but they cater to a very limited schematic of programs, and thus are limited in their applicability [15]. Schema-guided synthesis specialized to the arithmetic domain has been proposed using a constraint-based solving methodology [4]. While the specification of our program synthesis task is comparable to these approaches, satisfiability-based solving makes our technique more efficient.

There has been some more recent work that can be viewed as deductive synthesis. One proposal consists of refining the proof of correctness, simultaneously with the process of program refinement [61]. This is a novel take on deductive synthesis, but it is unclear whether the program can be efficiently found in this much larger search space. In the line of work treating synthesis as a search problem, decision procedures for program synthesis [39, 45] have also been proposed. Their expressivity is limited, i.e., they work for restricted classes of programs, but the efficiency of synthesis is more predictable.

Inductive Synthesis Inductive synthesis generalizes from instances to generate a program that explains all instances or traces that meet a specification. The instances could be *positive* ones that define valid behavior or counterexamples that eliminate invalid behavior.

Of particular note in this category is the work by Bodik and Solar-Lezama et al. on the Sketch system, which synthesizes from partial programs [55]. The Sketch system fills out integer holes, whose values may be difficult for the programmer to specify, in a partial program. On the other hand, our verification inspired approach infers and fills out holes with arbitrary expressions, or arbitrary predicate sets, as required. In particular, while holes in Sketch have finite values, in our approach they can be symbolic and thus arbitrary valued. Additionally, sketch only allows partial programs, while we have both partial program and partial invariants.

In Sketch, a model checker eliminates invalid candidate programs, by attempting to verify a candidate program that the synthesizer enumerates heuristically using a guided search. We never generate any candidate programs, but any solutions to our system of constraints is a valid synthesis solutions. Loops are handled incompletely in Sketch, by unrolling or by using a predefined skeleton, while there is precise invariant-based summarization in our approach.

Inductive synthesis has also been used for synthesizing programs in specialized domains such as loop-free programs [26,36] (e.g., bit-vector algorithms) and programs represented by simple logical formulas [35] (e.g., graph classifiers). These techniques also perform counterexample guided inductive synthesis using SAT/SMT solvers. However, the restriction to specialized domains allows searching for full programs as opposed to filling holes in partial programs or templates.

One of the biggest success stories of inductive synthesis has been in the domain of *end-user programming* including synthesis of string processing macros [25] and table layout transformations[33]. Unlike above-mentioned systems, which use SAT/SMT solvers, the key technique employed by these systems is to structurally reduce the synthesis problem to that of synthesizing simpler subexpressions. Another interesting application of inductive synthesis has been in the educational domain of ruler/compass based geometry constructions [27] in the context of building *intelligent tutoring systems*. The key technique used here is to prune exhaustive search by using AI-style heuristics.

8 Conclusion

Program verification consists of finding invariants that prove program correctness, and optionally inferring preconditions for correctness. Program synthesis consists of finding a program which meets a given specification. This paper describes program synthesis and program verification techniques based on *templates*. Templates are programmer-specified partial descriptions with holes. For program verification, the verifier completes templates into concrete invariants. For program synthesis, the synthesizer completes the templates into concrete programs.

We have found that templates enable two things. One, they allow the programmer to specify the invariants, and programs of interest, in the process restricting the search space and making verification and synthesis feasible for difficult domains as well. Two, they permit the use of satisfiability solvers as a mechanism for solving these problems. Satisfiability solvers have seen incredible engineering advances and leveraging them brings those advances to verification and synthesis.

Our experiments with using template-based techniques has shown promise for this approach in verifying and synthesizing programs that were outside the scope of previous approaches.

References

1. F. L. Bauer, H. Ehler, A. Horsch, B. Moeller, H. Partsch, O. Paukner, and P. Pepper. *The/Munich Project CIP*. 1988.
2. Ottmar Beucher. *MATLAB und Simulink (Scientific Computing)*. Pearson Studium, 08 2006.
3. Dirk Beyer, Tom Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, 2007.
4. Michael Colón. Schema-guided synthesis of imperative programs by constraint solving. In *LOPSTR*, pages 166–181, 2004.
5. Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.
6. R L Constable. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.
7. Byron Cook, Ashutosh Gupta, Stephen Magill, Andrey Rybalchenko, Jirí Simsa, Satnam Singh, and Viktor Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD*, pages 205–212, 2009.
8. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
9. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

10. Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.
11. Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD*, pages 19–32, 2002.
12. Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for smt solvers. In *CADE-21*, pages 183–198, 2007.
13. Leonardo de Moura and Nikolaj Bjørner. Z3, 2008. `http://research.microsoft.com/projects/Z3/`.
14. Ewen Denney. *A Theory of Program Refinement.* PhD thesis, University of Edinburgh, 1999.
15. Joe W. Duran. Heuristics for program synthesis using loop invariants. In *ACM*, pages 891–900, 1978.
16. Thomas Emerson and Mark H. Burstein. Development of a constraint-based airlift scheduler by program synthesis from formal specifications. In *ASE*, page 267, 1999.
17. Bernd Fischer and Johann Schumann. Autobayes: a system for generating data analysis programs from statistical models. *J. Funct. Program.*, 13(3):483–508, 2003.
18. Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.
19. Pierre Flener, Kung-Kiu Lau, Mario Ornaghi, and Julian Richardson. An abstract formalization of correct schemas for program synthesis. *J. Symb. Comput.*, 30(1):93–127, 2000.
20. Denis Gopan and Thomas W. Reps. Lookahead widening. In *CAV*, pages 452–466, 2006.
21. Denis Gopan and Thomas W. Reps. Guided static analysis. In *SAS*, pages 349–365, 2007.
22. Cordell Green. Application of theorem proving to problem solving. In *IJCAI*, pages 219–239, 1969.
23. Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically refining abstract interpretations. *TR-07-23*, (TR-07-23), 2007.
24. Sumit Gulwani. Dimensions in program synthesis. In *FMCAD*, page 1, 2010.
25. Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.
26. Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.
27. Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61, 2011.
28. Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
29. Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.
30. Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, 2009.
31. Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. In *TACAS*, pages 262–276, 2009.
32. Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *CAV*, pages 634–640, 2009.
33. William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.
34. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
35. Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46, 2010.
36. Susmit Jha, Sumit Gulwani, Sanjit Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.
37. Ranjit Jhala and Ken McMillan. Array abstractions from proofs. In *CAV*, 2007.
38. Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of extended ml: a gentle introduction. *Theor. Comput. Sci.*, 173:445–484, 1997.
39. Viktor Kuncak, Mikael Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, 2010.
40. Shuvendu K. Lahiri and Randal E. Bryant. Constructing quantified invariants via predicate abstraction. *VMCAI*, pages 331–353, 2004.
41. Shuvendu K. Lahiri and Randal E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.
42. Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Logic*, 9(1):4, 2007.
43. Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.
44. Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.
45. Mikael Mayer, Philippe Suter, Ruzica Piskac, and Viktor Kuncak. Comfusy: Complete functional synthesis (tool presentation). In *CAV*, 2010.
46. John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposia in Applied Mathematicas*, pages 33–41. American Mathematical Society, 1967.
47. James McDonald and John Anton. SPECWARE - producing software correct by construction. Technical Report KES.U.01.3., 2001.
48. J. M. Morris and A. Bunkenburg. Specificational functions. *ACM Trans. Program. Lang. Syst.*, 21:677–701, 1999.
49. Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.
50. Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Constraint-based linear-relations analysis. In *SAS*, pages 53–68, 2004.
51. Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, pages 25–41, 2005.
52. Alexandeer Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.
53. D. R. Smith. Kids: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16(9):1024–1043, 1990.

54. Douglas R. Smith. Designware: software development by refinement. pages 3–21, 2001.
55. Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.
56. Saurabh Srivastava. *Satisfiability-based Program Reasoning and Program Synthesis*. PhD thesis, University of Maryland, College Park, 2010. `http://www.cs.umd.edu/~saurabhs/pubs/saurabh-srivastava-thesis-9pt.pdf`.
57. Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, 2009.
58. Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. VS3: SMT solvers for program verification. In *CAV*, 2009.
59. Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, 2010.
60. William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC*, pages 179–196, 2002.
61. Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, 2010.
62. Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. In *IJCAI*, pages 241–252, 1969.
63. Chao Wang, Zijiang Yang, Aarti Gupta, and Franjo Ivancic. Using counterex. for improv. the prec. of reachability comput. with polyhedra. In *CAV*, pages 352–365, 2007.
64. Yichen Xie and Alexander Aiken. Saturn: A sat-based tool for bug detection. In *CAV*, pages 139–143, 2005.