Bugs, Moles and Skeletons: Symbolic Reasoning for Software Development

Leonardo de Moura and Nikolaj Bjørner

Microsoft Research, One Microsoft Way, Redmond, WA, 98052, USA
{leonardo, nbjorner}@microsoft.com

Abstract. Symbolic reasoning is in the core of many software development tools such as: bug-finders, test-case generators, and verifiers. Of renewed interest is the use of symbolic reasoning for synthesing code, loop invariants and ranking functions. Satisfiability Modulo Theories (SMT) solvers have been the focus of increased recent attention thanks to technological advances and an increasing number of applications. In this paper we review some of these applications that use software verifiers as bug-finders "on steroids" and suggest that new model finding techniques are needed to increase the set of applications supported by these solvers.

1 Introduction

Symbolic reasoning is present in many diverse areas including software and hardware verification, type inference, static program analysis, test-case generation, scheduling and planning. In the software industry, symbolic reasoning has been successfully used in many test-case generation and bug-finding tools.

Symbolic reasoning is attractive for *verifiation*, but we have found it even more compelling for finding **bugs**. Test-case generation tools produce **moles** that are test inputs which exercise particular program paths and their main goal is to increase code coverage. Of recent interest is the use of **skeletons**, also known as templates or schemas, when using symbolic reasoning in domain specific ways.

We claim these tools are successful in industry because their results, moles and bugs, can be easily digested, and domain specific skeletons are simple to formulate. For example, generated moles can be directly executed on the system under test. So it is straightforward to check and re-use the result from test-case generation tools. We here make a case for the importance of symbolic reasoners supporting the hunt for bugs and moles and the creation of skeletons.

A long-running and natural use of symbolic reasoning tools has been in the context of *program verification*, and indeed, a lot of our experience with symbolic reasoning has been rooted in program verification systems. The ideal of verified software has been a long-running quest since Floyd and Hoare introduced program verification by assigning logical assertions to programs. Yet, the starting point of this paper is making a case that using symbolic reasoning tools are compelling in the context of even partial program exploration and design, and

this domain offers compelling challenges for symbolic reasoning systems. The ideal of verified software amounts to a formidable task. It includes grasping with problems that are often quite tangentical to the software being verified. Common pittfalls are that an axiomatization of the environment/runtime may be incorrect and the properties being verified are not the right ones. Such pittfalls are hard to avoid as verification is intimately tied to abstraction. Unfortunately unsound abstractions are so much easier to come around than sound ones. Examples where unsound abstractions creep in include using arithmetic over the integers ($\mathcal{Z}$) instead of machine arithmetic, using memory model simplifications (e.g., pointer arithmetic), and ignoring concurrency. In other cases the challenge is not about simplifying the verification task, but it is about correctly encoding the underlying environment and runtime. Finally, with a Floyd-Hoare proof in house, and a trusted system model, the question is *how can I trust the verifier?* One approach to answering this question is by using *certificates* generated by the symbolic reasoning system. However, certificate generation can produce a significant overhead on automatic theorem provers in terms of memory and time. Another solution is the use of certified theorem provers. Verifying the verifier has become the ultimate distraction.

Independently of all these hurdles, in our point of view, software verification systems can be seen as bug-finding tools with *much better coverage*. Of course, in this case, the tool must be capable of reporting why a proof attempt did not succeeded. This view is used in almost every software verification project at Microsoft. It is not uncommon for these projects to demonstrate value by reporting the discovery of non-trivial bugs in software that was heavily tested by standard techniques. Following this view, a certificate/proof should be seen as a "the verifier cannot find more bugs for you" result.

Between these two extremes, bug-finding and verification, there is another application that is undergoing a rennaisance: synthesis. The idea of synthesizing code is not new, it dates back to the late 60's [12, 16]. Due to the recent advances in first-order theorem proving, SMT and QBF solving, it is becoming more feasible to synthesize non trivial glue code [15], small algorithms [17], ranking functions [4] and procedures [14]. The outcome of a synthesis tool is not as simple to check as the one produced by a bug-finding tool, but it is more tangible than a proof of correctness. In principle, developers can inspect and test the synthesized code independently of the symbolic reasoner.

2 Symbolic Reasoning at Microsoft

Z3 [5] is an SMT solver and the main symbolic reasoning engine used at Microsoft. SMT solvers combine the problem of Boolean Satisfiability with domains, such as, those studied in convex optimization and term-manipulating symbolic systems. They involve the decision problem, completeness and incompleteness of logical theories, and finally complexity theory.

2.1 Dynamic Symbolic Execution

SMT solvers play a central role in the context of *dynamic* symbolic execution, also called *smart white-box fuzzing*. There are today several industry applied tools based on dynamic symbolic execution, including CUTE, Exe, DART, SAGE, Pex, and Yogi [11]. These tools collect explored program paths as formulas and use solvers for identifying new test input (moles) that can steer execution into new branches.

SMT solvers are a good fit for symbolic execution because they rely on a solver that can find feasible solutions to logical constraints. They also use combinations of theories that are already supported by the solvers. To illustrate the basic idea of dynamic symbolic execution consider the greatest common divisor program 2.1. It takes the inputs x and y and produces the greatest common divisor of x and y.

```
int GCD(int x,int y) {
  while (true) {
    int m = x % y;
    if (m == 0) return y;
    x = y;
    y = m;
  }
}
```

Program 2.1: GCD Program

Program 2.2 represents the static single assignment unfolding corresponding to the case where the loop is exited in the second iteration. We use assertions to enforce that the condition of the if-statement is not satisfied in the first iteration, and it is in the second. The sequence of instructions is equivalently represented as a formula where the assignment statements have been turned into equations.

```
int GCD(int x0,int y0) {
  int m0 = x0 % y0;
  assert (m0 != 0);
  int x1 = y0;
  int y1 = m0;
  int m1 = x1 % y1;
  assert (m1 == 0);
}
```

$$
\begin{array}{l}
(m_0 = x_0 \ \% \ y_0) \ \wedge \\
\neg(m_0 = 0) \quad\quad \wedge \\
(x_1 = y_0) \quad\quad \wedge \\
(y_1 = m_0) \quad\quad \wedge \\
(m_1 = x_1 \ \% \ y_1) \ \wedge \\
(m_1 = 0)
\end{array}
$$

Program 2.2: GCD Path Formula

The resulting path formula is satisfiable. One satisfying assignment that can be found using an SMT solver is of the form:

$$x_0 = 2,\ y_0 = 4,\ m_0 = 2,\ x_1 = 4,\ y_1 = 2,\ m_1 = 0$$

It can be used as input to the original program. In the case of this example, the call `GCD(2,4)` causes the loop to be entered twice, as expected. Smart white-

box fuzzing is actively used at Microsoft. It complements traditional black-box fuzzing, where the program being fuzzed is opaque, and fuzzing is performed by pertubing input vectors using random walks. It has been instrumental in uncovering several subtle security critical bugs that black-box methods have been unable to find.

2.2 Static Program Analysis

Static program analysis tools work in a similar way as dynamic symbolic execution tools. They also check feasibility of program paths. On the other hand they can analyze software libraries and utilities independently of how they are used. One advantage of using modern SMT solvers in static program analysis is that SMT solvers nowadays accurately capture the semantics of most basic operations used by commonly used programming languages. We use the program in Figure 1 to illustrate the need for static program analysis to use bit-precise reasoning. The program searches for an index in a sorted array `arr` that contains a key.

```
int binary_search(
  int[] arr, int low, int high, int key) {
    assert (low > high || 0 <= low < high);
    while (low <= high) {
        // Find middle value
        int mid = (low + high) / 2;
        assert (0 <= mid < high);
        int val = arr[mid];
        // Refine range
        if (key == val) return mid;
        if (val > key) low = mid+1;
        else high = mid-1;
    }
    return -1;
}
```

Fig. 1. Binary search

The `assert` statement is a *pre-condition*, for the procedure. It restricts the input to fall within the bounds of the array `arr`. The program performs several operations involving arithmetic, so a theory and corresponding solver that understands arithmetic appears to be a good match. It is however important to take into account that languages, such as Java, C# and C/C++ all use 32-bit integers as the representation for values of type `int`. This means that the accurate theory for `int` is two-complements modular arithmetic. The maximal positive 32-bit integer is $2^{31} - 1$ and the smallest negative 32-bit integer is -2^{31}.

If both `low` and `high` are 2^{30}, `low + high` evaluates to 2^{31}, which is treated as the negative number -2^{31}. The presumed assertion $0 \leq mid < high$ therefore does not hold. Fortunately, several modern SMT solvers support the theory of *bit-vectors*, which accurately captures the semantics of modular arithmetic. The bug does not escape an analysis based on the theory of bit-vectors. Such an analysis would check that the array read `arr[mid]` is within bounds during the first iteration by checking the formula:

$$\begin{array}{ll} & low > high \vee 0 \leq low < high < arr.length \\ \wedge & low \leq high \\ \rightarrow & 0 \leq (low + high)/2 < arr.length \end{array}$$

As we saw, the formula is not valid. The values $\texttt{low} = \texttt{high} = 2^{30}, \texttt{arr.length} = 2^{30} + 1$ provide a counter-example. An integration with the solver Z3 and the static analysis tool PREfix led to the automatic discovery of several overflow-related bugs in Microsoft's rather large code-base.

2.3 Software Verification

Extended static checking uses the methods developed for program verification, but in the more limited context of checking absence of run-time errors. The SMT solver Simplify [7] was developed in the context of the extended static checking systems ESC/Modula 3 and ESC/Java [10]. This work has been the inspiration for several subsequent extended static program checkers, including Why [9] and Boogie [1]. These systems are actively used as bridges from several different front-ends to SMT solver backends. Boogie, for instance, is used as a backend for systems that verify code from languages, such as an extended version of C# (called Spec#), as well as low level systems code written in C. Current practice indicates that one person can drive these tools to verify selected extended static properties of large code bases with several hundreds of thousands of lines. This effort relies heavily on some of the automated methods used in software model-checking. A more ambitious project is the Verifying C-Compiler system [8], which targets functional correctness properties of Microsoft's Viridian Hyper-Visor. The Hyper-Visor is a relatively small (100K lines) operating system layer, yet correctness properties are challenging to formulate and establish. The entire verification effort is estimated to be around 60 man-years.

2.4 Synthesis

Finally, there is recent and active interest in using modern SMT solvers in the context of synthesis of inductive loop invariants [18] and synthesis of program fragments [14], such as sorting, matrix multiplication, de-compression, graph, and bit-manipulating algorithms. Take for instance the Strassen's matrix multiplication algorithm in the special case of 2×2 matrices. Synthesizing it amounts to arranging a set of (7) multipliers and adders to obtain equivalent results as the standard matrix multiplication algorithm that uses 8 multipliers. The search

process can be carried out on a multipliers that manipulate words of length 2-3 bits. The synthesized code can then be checked on full bit-widths (32 or 64 bits). These applications share a common trait in the way they use their underlying symbolic solver. They search a template *vocabulary* of instructions, that are composed as a model in a satisfying assignment. Section 3.3 goes into more detail.

3 Symbolic Reasoning Support for Models

3.1 Streams of Candidate Models

Most SMT solvers are capable of producing models for satisfiable quantifier-free formulas. A model is an interpretation that makes the formula true. For example, the interpretation $\{a \mapsto 2, b \mapsto 5\}$ is a model for the formula $a \geq 0 \wedge b \geq a + 3$. This capability is essential in many industrial applications, because moles and bugs are extracted from models.

Quantifiers are usually used to axiomatize the environment/runtime, state properties, specify frame axioms, etc. For example, the formula $\forall i, j.\ i \leq j \rightarrow f(i) \leq f(j)$ is used to say that f is a non-decreasing function. Quantifier reasoning in SMT is a long-standing challenge. The practical method employed in modern SMT solvers is to instantiate quantified formulas based on heuristics, which is not refutationally complete even for pure first-order logic. Moreover, refutationally complete procedures are not sufficient, since they will only guarantee that a proof of unsatisfiability will be found eventually for unsatisfiable formulas. However, in industry, we are mainly interested in the satisfiable instances, where a refutationally complete procedure may not even terminate. Some SMT solvers support decidable fragments [2, 6, 20], unfortunately they are not expressive enough to encode all symbolic reasoning problems found in practice.

A pragmatic approach for dealing with the problem above is to produce *candidate models*. Given a formula of the form $F \wedge G$, where G is quantifier-free, a candidate model is an interpretation that satisfies G and many instances of the universally quantified formulas in F. For example, consider the following simple satisfiable formula

$$\overbrace{\forall i, j.\ i \leq j \rightarrow f(i) \leq f(j)}^{F} \wedge \underbrace{w \geq v + 2 \ \wedge \ f(v) \leq 1 \ \wedge \ f(w) \leq 3}_{G}$$

Standard SMT solvers will produce a candidate model such as:

$$v \mapsto 0,\ w \mapsto 2,\ f \mapsto [0 \mapsto 1,\ 2 \mapsto 3,\ \mathit{else} \mapsto 4]$$

The interpretation for f is a *function graph*, it states that the value of the function f at 0 is 1, at 2 is 3, and for all other values is 4. This interpretation

satisfies G, and satisfies the instance $v \leq w \rightarrow f(v) \leq f(w)$ of F, but it clearly does not satisfy F.

Candidate models are relevant because they may contain enough information to help the developer to understand why some property does not hold, or some program location is reachable. Moles and bugs may still be extracted from them, and the actual program (i.e., the definitive oracle) can be executed to confirm they are indeed correct. This observation suggests a particular tool flow not very often explored. The basic idea is to use the actual program as an oracle, to check whether the candidate model really induces a valid mole/bug or not. If it does, then the tool terminates. Otherwise, it informs the solver that the candidate model is a not valid, and the search continues. In this approach, the solver is forced to generate a *stream* of more and more refined candidate models until a valid mole/bug can be successfully extracted.

3.2 Model Checking Quantifiers

Given a candidate model I, it is useful to have a procedure P that checks whether the interpretation I satisfies a universally quantified formula F or not. We say P is a *model checking procedure*. To describe how P can be constructed, let us describe how interpretations are particularly encoded in Z3. In Z3, we assume there is an intended interpretation $\mathcal{T}$ for the supported set of theories T. In the case of Z3, T is the union of the following theories: linear arithmetic, bit-vectors, arrays, inductive data-types, and uninterpreted functions. Given a satisfiable formula F, a model I is a function that maps the structure $\mathcal{T}$ that satisfies T, into an expanded structure M that satisfies $F \cup T$. Our models also come equipped with a set of formulas R that restricts the class of structures that satisfy T. For example, if T is the empty theory, then R is just a cardinality constraint on the size of the universe. When needed, we use fresh constant symbols k_1, $\ldots$, k_n (ur-elements) to name the elements in $|M|$ (i.e., the universe of M). In Z3, the interpretation of an uninterpreted symbol s is an expression $I_s[\bar{x}]$, which contains only interpreted symbols and the fresh constants k_1, $\ldots$, k_n. For uninterpreted constants c, $I_c[\bar{x}]$ is just a ground term I_c. For uninterpreted function and predicate symbols, the term $I_s[\bar{x}]$ should be viewed as a *lambda expression*. For example, the candidate model described in the previous section is encoded as:

$$v \mapsto 0,\ w \mapsto 2,\ f(x) \mapsto ite(x = 0, 1, ite(x = 2, 3, 4))$$

Where $ite(c, t, e)$ is the if-the-else term.

When models are encoded this way, it is straightforward to check whether a universally quantified formula $\forall \bar{x}.\ F[\bar{x}]$ is satisfied by a candidate model or not [20]. Let $F^I[\bar{x}]$ be the formula obtained from $F[\bar{x}]$ by replacing any term $f(\bar{t})$ with $I_f[\bar{t}]$, when f is uninterpreted. We claim a candidate model satisfies $\forall \bar{x}.\ F[\bar{x}]$ if and only if $R \wedge \neg F^I[\bar{s}]$ is unsatisfiable, where $\bar{s}$ is a tuple of fresh constant symbols. In the previous example, the formula $\forall i, j.\ i \leq j \rightarrow f(i) \leq f(j)$ is not satisfied by the candidate model, because the following formula is satisfiable.

$$s_1 \leq s_2 \wedge \neg(ite(s_1 = 0, 1, ite(s_1 = 2, 3, 4)) \leq ite(s_2 = 0, 1, ite(s_2 = 2, 3, 4)))$$

For instance, this formula is satisfied by $\{s_1 \mapsto 1, s_2 \mapsto 2\}$.

Similarly to the oracle-approach based on the actual program, new instances of universally quantified formulas can be extracted from failed model checking attempts. The new instance has the property that it will "block" the current candidate model from being produced again by the solver.

This particular way of encoding models allows Z3 to represent interpretations for function symbols that are not expressible by finite function graphs. For example, the following candidate model

$$v \mapsto 0,\ w \mapsto 2,\ f(x) \mapsto ite(x \leq 0, 1, ite(x \leq 2, 3, 4))$$

is a model for our working example, because the following ground formula is unsatisfiable.

$$s_1 \leq s_2 \wedge \neg(ite(s_1 \leq 0, 1, ite(s_1 \leq 2, 3, 4)) \leq ite(s_2 \leq 0, 1, ite(s_2 \leq 2, 3, 4)))$$

Candidate models with this particular shape can be automatically computed because our example is in the *array property* decidable fragment [2].

3.3 Skeleton Based Model Finding & Synthesis

Satisfiability solvers have been used to synthesize loop invariants [3, 13], code [19], and ranking functions [4]. To illustrate these ideas, consider the following abstract program:

pre
while (c) {
  T
}
post

In the loop invariant synthesis problem, we want to synthesize a predicate I that can be used to show that *post* holds in the end of the *while-loop*. Let, $pre[s]$ be a formula encoding the set of states reachable before the beginning of the loop, $c[s]$ be the encoding of the entering condition, $T[s, s']$ be the transition relation, and $post[s]$ be the encoding of the property we want to prove. Then, the loop invariant exists if the following formula is satisfiable, and any model can be used to extract the loop invariant.

$$\begin{array}{l}\forall s.\ pre[s] \rightarrow I(s) \wedge \\ \forall s, s'.\ I(s) \wedge c[s] \wedge T[s, s'] \rightarrow I(s') \wedge \\ \forall s.\ I(s) \wedge \neg c[s] \rightarrow post[s]\end{array}$$

Similarly, in the ranking function synthesis problem, we want to synthesize a function *rank* that decreases after each loop iteration. The idea is to use this function to show a particular loop always terminate in the program. This problem can be encoded as the following satisfiability problem.

$$\begin{array}{l}\forall s.\ rank(s) \geq 0 \wedge \\ \forall s, s'.\ c[s] \wedge T[s, s'] \rightarrow rank(s') < rank(s)\end{array}$$

Let us now illustrate these general schemas using the following simple example program. The program increments x and y in lock-step in a loop and we wish to check that the loop terminates and that $y = n$ at the end of the loop.

```
assert(n >= 0);
x = 0; y = 0;
while (x < n) {
  x = x + 1;
  y = y + 1;
}
assert(y == n);
```

For this simple program, the formulas associated with invariant and ranking synthesis problems are:

$$\begin{array}{l}\forall x, y, n.\ n \geq 0 \wedge x = 0 \wedge y = 0 \rightarrow I(x, y, n) \wedge \\ \forall x, y, n, x', y', n'.\ I(x, y, n) \wedge x < n \wedge x' = x + 1 \wedge y' = y + 1 \wedge n' = n \rightarrow \\ \qquad\qquad I(x', y', n') \wedge \\ \forall x, y, n.\ I(x, y, n) \wedge \neg(x < n) \rightarrow y = n\end{array}$$

and

$$\begin{array}{l}\forall x, y, n.\ \mathit{rank}(x, y, n) \geq 0 \wedge \\ \forall x, y, n, x', y', n'.\ x < n \wedge x' = x + 1 \wedge y' = y + 1 \wedge n' = n \rightarrow \\ \qquad\qquad \mathit{rank}(x', y', n') < \mathit{rank}(x, y, n)\end{array}$$

Both formulas are satisfiable, the following interpretations are models for them:

$$I(x, y) \mapsto x = y \wedge x \leq n$$

and

$$\mathit{rank}(x, y, n) \mapsto \mathit{ite}(x \leq n, n - x, 0)$$

Thus, in principle, these problems can be attacked by any SMT solver with support for universally quantified formulas, and capable of producing models. Unfortunately, to the best of our knowledge, no SMT solver can handle this kind of problem, even when n, x and y range over finite domains, such as machine integers. They will not terminate or give-up in both problems. For these reasons, many synthesis tools only use SMT solvers to decide quantifier-free formulas. In these applications, the SMT solver is usually used to check whether a candidate interpretation for I and *rank* is valid or not. The synthesis tool search for candidate interpretations using *templates*. Abstractly, a template is a **skeleton** that can be instantiated. For example, when searching for a ranking function, the synthesis tool may limit the search to functions that are linear combinations of the input.

This approach can be easily incorporated to SMT solvers that support the techniques described in the previous section. Given a collection of skeletons, the basic idea is to search for models where the intepretation of function and predicate symbols are instances of the given skeletons. We say an SMT solver based on this approach is a *skeleton based model finder*. In this context, an SMT solver may even report a formula to be *unsatisfiable modulo a collection of skeletons*.

Similarly to the approach used to represent models in Z3 (Section 3.2), skeletons are expressions containing free variables, and should be also viewed as

lambda expressions. However, skeletons may also contain fresh constants that must be instantiated. For example, the skeleton $ax + b$, where a and b are fresh constants, may be used as a template for the interpretation of unary function symbols. The expressions $x + 1$ ($\{a \mapsto 1, b \mapsto 1\}$) and $2x$ ($\{a \mapsto 2, b \mapsto 0\}$) are instances of this skeleton.

As usual, we assume the input formula is of the form $F \wedge G$, where G is quantifier free. We also assume a collection of skeletons S is provided by the user. First, we use an SMT solver to check whether $F \wedge G$ is satisfiable or not. If it returns *unsat* or *sat*, then we terminate. In practice, for satisfiable instances, the SMT solver will most likely return just a candidate model. Then, for each function symbol f in G, we select a skeleton $s_f[\bar{x}]$ from S. Next, we check whether the following formula is satisfiable or not.

$$F \wedge G \wedge \bigwedge_{f(\bar{t}) \in G} f(\bar{t}) = s_f[\bar{t}]$$

If this formula is unsatisfiable, we conclude that the selected skeletons cannot be used to satisfy the formula. Let C be the set of fresh constants used in the skeletons. So, if the SMT solver returns a candidate model, it must assign values for each constant in C, and these values are used to instantiate the skeletons. After the skeletons are instantiated, the new interpretation for the uninterpreted function symbols can be checked using the model checking technique described in Section 3.2. If the model checking step fails, then new quantifier instances are generated and added to G, and the process continues.

For example, consider the following very simple formula

$$\overbrace{\forall x.\ g(x) \geq 2x \ \wedge \ \forall x.\ f(x) \leq g(x) + 1}^{F} \wedge \underbrace{g(0) \leq 0 \ \wedge \ f(0) \geq 0}_{G}$$

Assume our collection of skeletons S is $\{ax + b\}$. That is, we are looking for models where the interpretation of every function symbol is a linear function. Assume the SMT solver terminates producing a candidate model, then we select $a_f x + b_f$ and $a_g x + b_g$ as the skeletons for f and g respectively. Note that we use a different set of fresh constants for f and g. Then, we check whether the following formula is satisfiable or not.

$$F \wedge G \wedge \underbrace{g(0) = a_g 0 + b_g \wedge f(0) = a_f 0 + b_f}_{E_1}$$

Assume the SMT solver returns a candidate model for the formula above assigning $\{a_g \mapsto 0, b_g \mapsto 0, a_f \mapsto 0, b_f \mapsto 0\}$. So, using this assignment, our interpretation for g and f is the constant function 0. This interpretation satisfies the quantifier $\forall x.\ f(x) \leq g(x) + 1$, but fails to satisfy $\forall x.\ g(x) \geq 2x$, because the induced model checking formula $\neg(0 \geq 2s_1)$ is satisfiable. A possible model

is $s_1 \mapsto 1$. Then, instantiating the quantifier with $x = 1$, we obtain a new set of ground formulas $G_1 = G \wedge g(1) \geq 2$. Assume the SMT solver terminates producing a candidate model for $F \wedge G_1$. Then, we check whether the following formula is satisfiable or not.

$$F \wedge G_1 \wedge \underbrace{E_1 \wedge g(1) = a_g + b_g}_{E_2}$$

In this case, we obtain the new assignment $\{a_g \mapsto 2, b_g \mapsto 0, a_f \mapsto 0, b_f \mapsto 0\}$, which corresponds to the interpretations $g(x) \mapsto 2x$ and $f(x) \mapsto 0$. Now, the first quantifier is satisfied, but $\forall x.\ f(x) \leq g(x) + 1$ fails because the model checking formula $\neg(0 \leq 2s_1 + 1)$ is satisfiable. A possible model is $s_1 \mapsto -1$. Then, we obtain $G_2 = G_1 \wedge f(-1) \leq g(-1) + 1$. Similarly to the previous steps, we check the satisfiability of

$$F \wedge G_2 \wedge E_2 \wedge g(-1) = -a_g + b_g \wedge f(-1) = -a_f + b_f$$

We obtain the assignment $\{a_g \mapsto 2, b_g \mapsto 0, a_f \mapsto 2, b_f \mapsto 0\}$, which corresponds to the interpretations $g(x) \mapsto 2x$ and $f(x) \mapsto 2x$. Now, both quantifiers are satisfied by this interpretation and the SMT solver can report $F \wedge G$ as satisfiable.

4 Conclusion

A long-running and natural use of symbolic reasoning tools has been in the context of *program verification*. However, given the many software verification and analysis tools used at Microsoft, we have found that the most attractive are the ones for finding bugs and producing moles. Skeletons enable a new set of promising applications based on synthesis. They are currently applied as layers on top of SMT solvers. We believe that supporting these techniques natively as part of quantifier instantiation engines is a useful and promising technique for searching models of quantified satisfiable formulas.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
2. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
3. M. Colón. Schema-guided synthesis of imperative programs by constraint solving. In Sandro Etalle, editor, *LOPSTR*, volume 3573 of *LNCS*, pages 166–181. Springer, 2004.

4. B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *TACAS*, volume 6015 of *LNCS*, pages 236–250. Springer, 2010.
5. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*. Springer, 2008.
6. L. de Moura and N. Bjørner. Deciding Effectively Propositional Logic using DPLL and substitution sets. In Allesandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, 2008.
7. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
8. E. Cohen and M. Dahlweid and M. Hillebrand and D. Leinenbach and M. Moskal and T. Santen and W. Schulte and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *TPHOL*, 2009.
9. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, 2003.
10. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *PLDI*, pages 234–245, 2002.
11. P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, 2008.
12. C. Cordell Green. Application of theorem proving to problem solving. In *IJCAI*, pages 219–240, 1969.
13. S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, 2009.
14. S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010 (to-appear).
15. M. R. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *ISMIS*, pages 326–335, 1994.
16. Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.
17. A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.
18. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *PDLI*, 2009.
19. S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. In *POPL*, 2010.
20. Y. Ge and L. de Moura. Complete instantiation for quantified SMT formulas. In *CAV*, 2009.