Nikolaj Bjørner Leonardo de Moura

Applications of SMT solvers to Program Verification

Rough notes for SSFT 2014
Prepared as part of a forthcoming revision of
Daniel Kröning and Ofer Strichman's book on
Decision Procedures

May 19, 2014

Springer

Contents

1

Applications of SMT Solvers

1.1 Introduction

A significant application domain for SMT solvers is in the analysis, verification, testing and construction of programs. This chapter covers some of these areas where SMT solvers have been used. Several of our applications are in the context of the Z3 SMT solver available from Microsoft Research, and several of the application areas we cover are based on uses of Z3. Our description, however, abstracts from implementation features. It provides an introduction to general SMT concepts and techniques that are of relevance for the applications we discuss.

The set of applications we will describe here can be roughly divided into two categories. One set of methods take as starting point a set of *properties* of interest uses these to control the interaction with the SMT engine. Program verification or verification of selected safety properties of programs is largely property oriented. Program verification systems typically require strengthening the properties of interest by introducing **inductive invariants** and additional pre- and post-conditions for procedures in order to check the original specification. The second category takes as a starting point program *executions* and uses an (abstract) execution trace as the basis for interacting with the SMT solver. Test-case generation tools are typically execution-oriented. They search over an under-approximation of the set of possible program executions. Symbolic program model checking tools are characterized by building symbolic representations of the reachable program states. These symbolic representations can be formulas that represent an over-approximation of reachable program states.

Naturally, there is not always a clear distinction between property and execution-guided analysis, and there are several tools that combine methodologies from the two points of view. From the perspective of the SMT solver, the two angles give rise to very different sets of verification challenges. The property-oriented analysis methods that we examine generate first-order formulas using quantifiers, and handling quantifier reasoning is critical. It is im-

portant to be able to establish proofs for these formulas. On the other hand, execution-guided systems generate quantifier-free conjunctive formulas, that can be large. It is important to be able to produce *models* for these formulas. The examples we present in this chapter illustrate these main points.

The range of applications we examine also illustrate the use of SMT solvers for both non-formal and formal verification. In the case of non-formal vrification, the main purpose is to find bugs. Model-finding capabilities are crucial in these cases. In the case of formal verification, the main application is to formally verify functional correctness of programs.

This chapter examines five application areas of SMT solvers. We start with describing **Dynamic Symbolic Execution** in Section 1.3. It illustrates using SMT solvers for checking path feasibility. **Symbolic Software Model Checking**, discussed in Section 1.4, use SMT solvers to bridge the gap from programs with large or unbounded state spaces into small finite state abstractions. We discuss uses of SMT solvers for scalable **static analysis** in Section 1.5, and finally Section 1.6 illustrates the bridge from functional program verification to SMT solvers. One way to classify the applications is how they trade scale for expressiveness and precision. In one end of the spectrum, program verification systems can be used to verify complex functional properties of programs, and on the other end static analysis tools aim to be scalable and target a more restricted set of properties.

We use a common format for illustrating the applications. It is based on a substrate of programs written in a mild variant of the C programming language. We start by describing this substrate.

1.2 From Programs to Logic

The assignment of logical meaning to programs dates back to seminal work of Floyd [12] and Hoare [16]. Many refinements and enhancements have been developed over the years. Connections between logic and programs have been used in the context of program verification, test-case generation and static analysis. The current state-of-the-art includes program verification systems that have been used to verify thousands of lines of industrial code. They include symbolic test-case generation systems that have been used to analyze hundreds of thousands of programs. Finally, static analysis systems have been used on code bases of millions of lines of code.

1.2.1 An Imperative Programming Language Substrate

We illustrate some of these basic ideas by using programs written using a strongly typed imperative language that corresponds to a strongly typed subset of C, with minor modifications to C++, Java and C#. Our examples only use a select set of features and we explain just these features.

We use three primitive types: `int` of signed integers, `unsigned int` of unsigned integers, and `bool` of Booleans. Values of type `int` are represented in twos complement 32-bit arithmetic. Thus, the range of signed integers is $[-2^{31}..2^{31}-1]$ and the range of unsigned integers is $[0..2^{32}-1]$. Arithmetical operations that produce integers outside of these ranges are said to *overflow*. The two Boolean values are `true` and `false`. It will be convenient to distinguish arrays and pointer types because we will be using the length of allocated arrays in programs and formulas. Pointers to values of type τ are denoted $\tau*$ and arrays of type τ are denoted $\tau[]$.

Expressions are formed from program variables and use side-effect-free operations, such as arithmetical $+, *, -$, equality $==$, disequality $!=$, logical conjunction $\&\&$, disjunction $||$, and negation $!$. Pointer dereferencing is written $*p$, and accessing an array a at position i is written $a[i]$. Expressions can be directly represented as first-order terms and formulas. For instance, pointer dereferencing $*p$ can be represented as the term $\texttt{deref}(h, p)$, where h tracks the state of the underlying program heap, and array access $a[i]$ by $\texttt{select}(a, i)$.

It will be convenient to use arrays that carry information about their lengths. So we will assume we can access the length of an allocated array a as `length(`a`)`, which we abbreviate as $|a|$. In the C programming language we would represent such arrays as a `struct` that contains a length field and the actual array. So in C, $\tau[]$ is shorthand for `struct` τ_A `{ int length; `τ`* contents; }`, and everytime we write $a[i]$ for an array a we mean a.`contents`$[i]$. In C# and Java, arrays are just arrays because the languages support a length function of the form a.`Length`(). Some of our examples allocate arrays. We use `alloc_array(`τ`, N)` to allocate arrays of type τ and with N elements. In the C programming language `alloc_array(`τ`, N)` is shorthand for $\{N, (\tau*)\texttt{malloc}(\texttt{sizeof}(\tau)*\texttt{N})\}$, and in Java and C#, `alloc_array(`τ`, N)` is shorthand for `new` τ`[N]`. The first element is accessed at position 0 and the last element of the allocated array is accessed at position $N-1$. We use the special constant `null` for pointers and arrays that are set to a special invalid value. Memory allocation may fail to allocate the requested resource. In this case `alloc_array` returns a value `null`. We use the predicate `is_allooc_array(a)` to state that pointers or arrays occupy allocated memory. Successful memory allocation ensures that `is_alloc_array` holds. To capture that allocated memory (of an array) is also initialized with values we use the predicate `init(a)`.

Statements comprise of the standard constructs known from C, C# and Java. An assignment of the value of expression e to program variable x is written $x = e$, a call to the procedure p using the arguments x and y is written $p(x, y)$, and we include conditional statments $\textbf{if}(e)S$; **else** S', while loops and returns. In addition, it will be convenient to also treat as primitive some additional statements. The **assert** e statement checks if the Boolean expression e is `true`. In an operational interpretation, execution aborts in an error state if e is false. If e is true, then execution proceeds unaffected. The **assume** e statement also checks e. Similar to **assert**, execution proceeds if e

is true, but in contrast execution aborts in a *success* state if e is false. Our use of **assert** and **assume** will be limited to expressions that do not have side-effects. For example, the assert and assume statements will not contain procedure calls that update the program state. The **havoc** x, y, z statement sets the values of program variables x, y, z to an arbitrary value.

1.2.2 Programs that are logic in disguise

The methods for assigning logical meaning to programs pioneered by Floyd and Hoare apply to programs with side-effects, loops and procedure calls. In general, imperative programs are not directly representable as first-order logical formulas. The main obstacle is the presence of unbounded loops and recursion. The obstacle is avoided if we restrict ourselves to programs with bounded recursion and bounded loops. Bounded recursion means that the depth of recursive calls is bounded by a fixed number. Likewise, bounded loops are traversed by a fixed number of iterations. In this case there is a straight-forward mapping from programs to logical formulas that summarize the input-output relations of the program. The mapping uses the *single static assignment* (SSA) representation of programs. The main idea is to unfold bounded loops and procedure calls and then create a time-stamped version of program variables, such that repeated assignments to the same program variable is tracked as assignments to different variables. Once the program is in SSA form, we can extract a logical formula by treating each assignment as an equality.

Let us illustrate the idea with an example.

Example 1.1. Consider Program 1.2.1.

Program 1.2.1 A recursion-free program with bounded loops and an SSA unfolding.

```
int Main(int x, int y)
{
  if (x < y)
     x = x + y;
  for (int i = 0; i < 3; ++i) {
      y = x + Next(y);
  }
  return x + y;
}

int Next(int x) {
  return x + 1;
}
```

```
int Main(int x0, int y0)
{
  int x1;
  if (x0 < y0)
     x1 = x0 + y0;
  else
     x1 = x0;
  int y1 = x1 + y0 + 1;
  int y2 = x1 + y1 + 1;
  int y3 = x1 + y2 + 1;
  return x1 + y3;
}
```

The initialization, the increments, and the loop test involving `i` have been inlined in the unfolding. The first line in the main program is an `if` statement that updates `x` if the condition holds. In the SSA form, we ensure that `x` is not updated in the case the condition does not hold. The input-output relation of the SSA program is now simple to express: Each assignment corresponds to an equality assertion, and conditional branch statements become conditional formulas.

$$\exists x_1, y_1, y_2, y_3 \left(\begin{array}{l} (x_0 < y_0 \implies x_1 = x_0 + y_0) \ \wedge \ (\neg(x_0 < y_0) \implies x_1 = x_0) \ \wedge \\ y_1 = x_1 + y_0 + 1 \ \wedge \ y_2 = x_1 + y_1 + 1 \ \wedge \ y_3 = x_1 + y_2 + 1 \ \wedge \\ \mathit{result} = x_1 + y_3 \end{array} \right)$$

1.3 Test-case Generation using Dynamic Symbolic Execution

1.3.1 The Application

Larger well-structured programs are typically composed from many smaller functions. Most of such functions can be treated as *units* that can be composed in different ways within other units. A *unit test* exercises one of these functions. The main problem in unit testing is to find input values to these functions that will exercise all interesting behaviors of the function and, during development and testing, uncover bugs in the unit. The problem of finding suitable input values that exercise all interesting behaviors is also relevant to whole programs (that depend on inputs). There are several approaches to enumerate a space of hopefully interesting inputs to a program. These range from generation of random test input to *fuzzing*. The starting point of fuzzers is a set of existing representative inputs to a program or unit. Fuzzers pertub the input in order to exercise different and hopefully useful variants. These approaches can quickly produce a large set of potentially useful inputs, but there is little to ensure that all the corner cases are covered.

The idea of symbolic simulation is to represent an execution path by the set of instructions that are executed during the path. These instructions can be converted into formulas: assignments are conjoined as equations and conditions from if-then-else statements are simply conjoined (with negation of the else-branch is taken). The satisfying instances of this formula correspond to input values that steer executions along this path. Other paths can be explored by negating one of the branch conditions encountered on the path. How is the execution path chosen? In symbolic simulation, the execution path can be chosen by extending a partial path with new conjuncts as long as the resulting formula is satisfiable. Dynamic symbolic simulation uses a concrete execution to guide the selection of paths: given the instructions executed on a concrete execution path, create a formula that corresponds to instructions up to a selected branch. The selected branch condition is added with the opposite sign (negation) as the concrete execution would produce. If the resulting

path formula is satisfiable, then a satisfying model for the formula produces new inputs that can guide execution to follow the opposite branch. Dynamic symbolic simulation also comes with the advantage that concrete executions come with concrete input values for a feasible path. Some of these values can be reused in cases where the program being analyzed makes a system call.

1.3.2 An Example

Program listing 1.3.1 contains the definition of a procedure `ReadBlocks`. The program is artificial, but it exemplifies low-level parsing code that is typically used when processing file formats, such as the media formats `jpeg`, `avi` or `mpeg`.

Program 1.3.1 Reading blocks from an array.

```
1  void ReadBlocks(int[] data, int cookie)
2  {
3    int i = 0;
4    while (true)
5    {
6      int next = data[i];
7      if (!(i < next && next < length(data))) return;
8      i = i + 1;
9      for (; i < next; i = i + 1) {
10         if (data[i] == cookie) {
11           i = i + 1;
12         }
13         else {
14           Process(data[i]);
15         }
16     }
17   }
18 }
```

The input array `data` encodes implicitly a linked list of blocks. Each block is linked by a `next` pointer that is an offset read from the `data` array. Reading `data` at offsets outside of the range `0..length(data)-1` results in a runtime error and we wish to check that the program does not encounter such errors.

The example intentionally contains an array access bug. When `i = next - 1` and `next = length(data) - 1` and `data[i] == cookie`, then `i` is incremented twice to `next + 1 = length(data)`. So the array access in line 6 violates the array bounds. We will here walk through the process that symbolic execution uses to identify this bug.

1.3.3 Methodolgy

Consider the execution trace that corresponds to running through the loop once and then exiting during the second iteration. Program listing 1.3.2 reproduces the sequence of instructions that corresponds to this execution. Each branch condition and runtime check (that is implicit in the original program listing) is included as an assertion: the successful execution along this path requires that each assertion holds. We have included line numbers from the original program to make it easier to trace back where the instructions come from.

Program 1.3.2 Constraints along a loop executed once.

```
3.  i = 0;
6.  assume data != null && 0 <= i && i < |data|;
6.  next = data[i];
7.  assume i < next && next < |data|;
8.  i = i + 1;
9.  assume i < next;
10. assume data != null && 0 <= i && i < |data|;
10. assume (data[i] != cookie);
14. assume data != null && 0 <= i && i < |data|;
14. Process(data[i]);
9.  i = i + 1;
9.  assume !(i < next);
6.  assume data != null && 0 <= i && i < |data|;
6.  next = data[i];
7.  assume !(i < next && next < |data|);
```

The trace does not contain any unbounded loops or recursive procedure calls. We can therefore use the SSA conversion as a step to obtain a formula that characterizes all values of the inputs `data` and `cookie` that exercise the same trace. We will here ignore the call to the procedure `Process`, which we here assume does not change the values of `data`. Of course, this is a simplifying assumption. Functions calling ReadBlocks could store the array `data` in a global variable that can be modified from `Process`.

The formula obtained from the SSA conversion is:

$$\exists i_0, i_1, i_2, \texttt{next}_0, \texttt{next}_1 \; . \; \begin{pmatrix} i_0 = 0 \wedge \\ \texttt{data} \neq \texttt{null} \; \wedge \; 0 \leq i_0 < |\texttt{data}| \wedge \\ \texttt{next}_0 = \texttt{data}[i_0] \wedge \\ (i_0 < \texttt{next}_0 \; \wedge \; \texttt{next}_0 < |\texttt{data}|) \wedge \\ i_1 = i_0 + 1 \wedge \\ i_1 < \texttt{next}_0 \wedge \\ \texttt{data} \neq \texttt{null} \; \wedge \; 0 \leq i_1 < |\texttt{data}| \wedge \\ \texttt{data}[i_1] \neq \texttt{cookie} \wedge \\ i_2 = i_1 + 1 \wedge \\ \neg(i_2 < \texttt{next}_0) \wedge \\ \texttt{data} \neq \texttt{null} \; \wedge \; 0 \leq i_2 < |\texttt{data}| \wedge \\ \texttt{next}_1 = \texttt{data}[i_2] \wedge \\ \neg(i_2 < \texttt{next}_1 \; \wedge \; \texttt{next}_1 < |\texttt{data}|) \end{pmatrix} \; . \tag{1.1}$$

The program variables that are local to `ReadBlocks` are existentially quantified and the input parameters `data` and `cookie` are free. It is simple to eliminate the existential quantifier: every existentially quantified variable occurs on the left hand side of an equation and there is no cyclic dependencies between quantified variables. So existential quantification can be eliminated by substituting the effect of assignments into the places where the updated values are used. The resulting formula is:

$$\begin{aligned} & \texttt{data} \neq \texttt{null} \; \wedge \; 0 \leq 0 < |\texttt{data}| \\ \wedge \; & 0 < \texttt{data}[0] \; \wedge \; \texttt{data}[0] < |\texttt{data}| \\ \wedge \; & 1 < \texttt{data}[0] \\ \wedge \; & \texttt{data} \neq \texttt{null} \; \wedge \; 1 \leq 1 < |\texttt{data}| \\ \wedge \; & \texttt{data}[1] \neq \texttt{cookie} \\ \wedge \; & \texttt{data} \neq \texttt{null} \; \wedge \; 0 \leq 2 < |\texttt{data}| \\ \wedge \; & \neg(2 < \texttt{data}[2] \; \wedge \; \texttt{data}[2] < |\texttt{data}|) \; . \end{aligned} \tag{1.2}$$

It follows that all interpretations of `data` and `cookie` that satisfy the formula exhibit the same path.

An exhaustive test-case generation process examines all possible branches of a program. This includes checking the negation of all branch conditions on the path. We then see that the path constraint

$$\texttt{data} \neq \texttt{null} \; \wedge \; \neg(0 \leq 0 < |\texttt{data}|)$$

is satisfied when `data` is an array of length 0. This input could cause the program to produce an access violation. This corner case is not too hard to encounter during random or fuzz testing. On the other hand the program has a different bug that requires a more sophisticated analysis. There is a possible array bounds violation when the last element in a range matches the `cookie` argument. In this case, the program variable i gets incremented past `next`. If in addition `next` points to the last element of `data` (that is, `next = |data| - 1`), then the subsequent access to `data[i]` reads past the bounds of `data`. The relevant path constraint is the satisfiable formula:

$$\begin{array}{l} \quad \texttt{data} \neq \texttt{null} \ \wedge \ 0 \leq 0 < |\texttt{data}| \\ \wedge\ 0 < \texttt{data}[0] \ \wedge \ \texttt{data}[0] < |\texttt{data}| \\ \wedge\ 1 < \texttt{data}[0] \\ \wedge\ \texttt{data} \neq \texttt{null} \ \wedge \ 1 \leq 1 < |\texttt{data}| \\ \wedge\ \texttt{data}[1] = \texttt{cookie} \\ \wedge\ \texttt{data} \neq \texttt{null} \ \wedge \ 0 \leq 2 \ \wedge \neg(2 < |\texttt{data}|) \ . \end{array} \tag{1.3}$$

1.3.4 Interfacing with SMT Solvers

The example illustrates the constraints produced as a byproduct of symbolic execution. We see that they are mainly a conjunction of atomic formulas. It is important to extract a satisfying assignment (a model) for these conjunctions. The model provides the test inputs that steer execution in new directions. Many path queries posed by symbolic execution engines can be highly-related. Consider for instance the path in Program 1.3.2. We are interested in finding input values that exercise different paths. Each assumption along this path represents a possible execution path that exercises a different branch. The different execution path is reached if we can find input to the program that satisfies the negated assumption together with the prefix of assumptions from the original path. There is something naturally incremental about this description: the assumptions along the path that was exercised are reused when probing further down in the path. On the other hand, the search has to consider both variants of each assumption: the negated version and the original version of the assumption as it appears on the path. This style of probing is suppored directly by modern SMT solvers. They contain interfaces for incrementally adding and retracting assertions using a stack discipline. The interfaces expose an operation, *push* that creates a new scope. All formulas that are asserted within this scope are retracted by a matching *pop* that exits the scope. Scopes can be nested. The concept is similar to the state updates made during backtracking DPLL search.

Let us illustrate how one can interact with SMT solvers using the standardized SMT-LIB2 textual interface [1]. It follows LISP conventions for the syntax so, for example, the equality formula `i = 0` is written `(= i 0)`. It contains the basic commands `assert` that takes a formula and conjoins it to the current logical context. The command `push` creates a scope, that is exited when there is a matching `pop` command. The `check-sat` command checks satisfiability of the logical context (everything that is asserted under the current set of pushes) and `get-model` obtains the satisfying assignment to the current set of variables. The illustration below simulates a search through different paths of Program 1.3.1. The first two `check-sat` invocations perform array bounds checks that are implicit with the access `data[i]` in line 6. It then pushes constraints that correspond to further executing the loop. Two branches are explored, the first where the loop condition `i < next` is false, and the other where it is true.

[1] `http://smtlib.org`

```
(set-option :model true)
(declare-fun i () Int)
(declare-fun i1 () Int)
(declare-fun next () Int)
(declare-fun data () (Array Int Int))
(declare-fun null () (Array Int Int))
(declare-fun length ((Array Int Int)) Int)

(assert (= i 0))
(push)
(assert (= data null))
(check-sat)
(get-model)
(pop)
(assert (not (= data null)))
(push)
(assert (not (and (<= 0 i) (< i (length data)))))
(check-sat)
(get-model)
(pop)
(assert (and (<= 0 i) (< i (length data))))
(check-sat)
(assert (= next (select data i)))
(assert (and (< i next) (< next (length data))))
(assert (= i1 (+ i 1)))
(push)
(assert (not (< i1 next)))
(check-sat)
(get-model)
(pop)
(assert (< i1 next))
(check-sat)
(get-model)
...
```

1.3.5 Industrial Adoption

Symbolic execution has recently found significant industrial adoption in the context of security analysis. The SAGE [13] tool that is based on dynamic symbolic execution is used to find most of the bugs identified by Microsoft's fuzz-testing infrastructure. It has been used on hundreds of parsers for various media formats and is administrated in a data-center test environment. The Klee tool [4], similarly, has been instrumental in finding a large number of security vunerabilities in code deployed on Windows and Linux. Finally, the Pex [13] tool offers an integration of dynamic symbolic execution with the

.NET runtime so that it can be used on any .NET language, including C#. Pex lets programmers directly take advantage of the symbolic execution technology for generating test inputs to .NET code. It offers a sophisticated integration with the .NET type system that enables it to generate test cases for complex structured data. Dynamic symbolic execution remains of interest also for the security community including "white", "blue" and "black" "hats" (jargon for industrial, legitimate and hackers with a shady purpose).

1.4 Symbolic Software Model checking

1.4.1 The Application

Dynamic symbolic execution finds some input that can guide execution into bugs. This method alone does not produce any guarantee that programs are free of all of the errors being checked for. The goal of *program model checking* tools is to automatically check a functional specification. The basic idea of program model-checking is to explore all possible executions using a finite and sufficiently small abstraction of the program state space.

1.4.2 An Example

We will use the program fragment in Program 1.4.1 as an example of producing a small finite state abstraction. It accesses requests using `GetNextRequest`. The call is protected by a lock to allow multiple threads to access the queue data-structure where requests are stored. Once a request is dequeued and released we can release the lock because the data associated with a request is not accessed by different threads. It is also important to release the lock before processing the request, which can take a long time or acquire different locks. It should not be possible to exit the loop without owning the lock. If it were, then the call to `unlock()` after the loop would release the lock twice. This violates how locks may be used: `unlock()` should never be called by a thread without that thread having acquired the lock using a call to `lock()`. The program has a very large number of states since the value of `count` can grow arbitrarily.

Program 1.4.1 Processing requests using locks.

```
1    do {
2          lock();
3          old_count = count;
4          request = GetNextRequest();
5          if (request != NULL) {
6              ReleaseRequest(request);
7              unlock();
8              ProcessRequest(request);
9              count = count + 1;
10         }
11   }
12   while (old_count != count);
13   unlock();
```

1.4.3 Methodolgy

From the point of view of checking correct uses of locks, the actual values of `count` and `old_count` are not important. On the other hand, the *relationship* between them contains useful information. Program 1.4.2 shows a finite state abstraction of the same locking program. The Boolean variable `b` encodes the relation `count == old_count`. We call this abstract program a *Boolean program*; the only type is Booleans. It is obtained from the original program by a method called *predicate abstraction* [14]. In general, a predicate abstraction may use many predicates to capture the behaviors of a program that are relevant to checking a specification. In this example a single predicate suffices. Given the finite Boolean program we can now explore the finite number of states to verify that the lock is always held when exiting the loop.

Program 1.4.2 Processing requests using locks, abstracted.

```
1   do {
2         lock();
3         b = true;
4         if (*) {
5            unlock();
6            if (b) {
7               b = false;
8            }
9            else {
10             havoc b;
11           }
12        }
13  }
14  while (!b);
15  unlock();
```

The finite state abstraction provided in Program 1.4.2 can be constructed from Program 1.4.1 by solving several logic queries. The approach we outline here abstracts each statemet in the program individually. For example, let us consider the statement `count = count + 1`. Let us examine the effect of this assignment on the predicate b : `count == old_count`. The predicate b' holds the value of b after an atomic statement. For example, in the case of the assignment `count = count + 1`, the variable b' stands for `count + 1 == old_count`. What relationships are there between b and b'? We can find these relationships by enumerating formulas using b and b' and check each formula for validity. For example the formula:

$$b \implies \neg b' : \ (\texttt{count = old\_count}) \implies \neg(\texttt{count+1 = old\_count})$$

is valid. The formula says that if the current value of `b` is **true**, then after executing the statement `count = count + 1` it will be **false**. Note that if `b` is **false**, then neither of the following conjectures is valid.

$$\neg(\texttt{count = old\_count}) \implies (\texttt{count+1 = old\_count})$$

$$\neg(\texttt{count = old\_count}) \implies \neg(\texttt{count+1 = old\_count})$$

In both cases, an SMT solver produces a counter-example to the conjecture. So when the current value of `b` is **false**, nothing can be said about its value after the execution of the statement. The result of these three proof attempts is then used to replace the statement

`count = count + 1;` by **if** `(b)` {`b =` **false;**} **else** {**havoc** `b;`}

Similarly, we replace the assignment `old_count = count;` by `b = true;` because b', which summarizes the effect of the assignment, is equivalent to the valid equality `count = count`. Furthermore, the loop test `count` $\neq$ `old_count` is just substituted with $\neg b$. Finally, the calls to `GetNextRequest`, `ReleaseRequest`, and `ProcessRequest` can be abstracted. The only property that is relevant with respect to checking locking is the test `request != NULL`, which we can replace by a non-deterministic branch.

This analysis allows forming Program 1.4.2, which now uses only a small number of finite states. A finite state model checker can now be used on the Boolean program. It will establish that `b` is always **true**, and that the lock is held, when control reaches calls to `unlock()`.

1.4.4 Interfacing with SMT Solvers

The example used just one predicate b and the method suggested to replace atomic statements by logical formulas over b and b'. There are 16 non-equivalent formulas using b and b'. In general, with $2n$ predicates $b_1, \ldots, b_n, b'_1, \ldots, b'_n$, there are $2^{2^{2n}}$ non-equivalent logical formulas. It is therefore highly infeasible to first enumerate these formulas and then check for validity. Many optimizations and heuristics are therefore used in tools for predicate abstraction. One approach searches for just valid implications of the form $\ell_1 \wedge \ldots \wedge \ell_n \implies b'$, where ℓ_i is either b_i or $\neg b_i$. This reduces the search space of possible abstractions; there are "only" $n2^n$ implication checks. Furthermore, *unsatisfiable cores* can be used to futher prune the set of redundant implications: if $b_1 \wedge b_3 \implies b'_2$ is valid, then both $b_1 \wedge b_2 \wedge b_3 \wedge \ldots \wedge b_{20} \implies b'_2$ and $b_1 \wedge \neg b_2 \wedge b_3 \wedge \ldots \wedge b_{20} \implies b'_2$ are valid. Recall that the basic idea behind using unsatisfiable cores is to extract a subset of assertions that were used. When checking $b_1 \wedge b_2 \wedge b_3 \wedge \ldots \wedge b_{20} \implies b'_2$ for validity, SMT solvers, dually check $b_1 \wedge b_2 \wedge b_3 \wedge \ldots \wedge b_{20} \wedge \neg b'_2$ for unsatsifiability. The set $b_1 \wedge b_3 \wedge \neg b'_2$ is already unsatisfiable, and SAT/SMT solvers can extract the unsatisfiable core. It is now redundant to enumerate other implications that contain b_1 and b_3.

1.4.5 Industrial Adoption

The approach to symbolic model checking of software we described here is used in the SDV [1] model checker that ships with Windows Server as part of the Driver Development Kit. SDV is used to model-check device driver software. Device driver software is low-level systems code that interacts with devices and the core operating system. The many interaction scenarios and assumptions makes the construction of such programs notoriously difficult. On the other hand, a faulty device driver can crash the entire operating system. The Symbolic Model Verifier tool SMV from Cadence [2] can also be used for

[2] `http://www.kenmcmil.com/`

software model checking. It uses an entirely different technique for finding a finite state abstraction. Instead of mining the program for suitable predicates to use for the abstraction it finds them by constructing Craig interpolants. It is beyond the scope of this chapter to explain the details of this method, but let us mention that this technique integrates with the explanations (proofs) produced by SMT solvers. The BLAST [15] tool has over time incorportated and developed techniques that relate to both predicate abstraction and interpolation. Many other software model checking tools that implement different techniques are being adapted in practice, for instance the Yogi [22] tool builds an abstract transition graph and refines it by propagating weakest preconditions. It is used as part of SDV as an independent model checker.

1.5 Static Analysis

1.5.1 The Application

Large scale industrial software is rarely executable as a stand-alone module with simple input and output behavior. System software, such as an operating system scheduler, a file system and associated filter drivers, a network stack and higher level network services are *reactive* as they maintain an ongoing interaction with an environment. Detailed finite state abstraction techniques, such as the one described in Section 1.4, have beem successful on device driver software, but have so far not been developed for fully automatic analysis of large scale software. Dynamic symbolic simulation techniques, described in Section 1.3, face the challenge of isolating or simulating environment interactions. Modular static analysis addresses the challenge with scalability and environment interactions by performing symbolic analysis of each procedure in isolation. Each analysis produces a **procedure summary**, that captures the behaviors of each procedure in a succinct way. The summaries are used when analyzing other procedures.

Typically, static analyzers take as starting point a program and a set of predefined properties, such as absence of errors division by zero, array bounds violations, heap access violations, and memory leaks. They then establish statically that no such errors are encountered, or they identify an abstract execution trace that can potentially cause a runtime error. Powerful static analysis engines are inter-procedural; they analyze the behavior of a given procedure using a compositional analysis of functions that call and are called from the analyzed procedure.

The PREfix [3] analysis tool pioneered a bottom-up analysis of procedures: it summarizes basic procedures as a set of guarded transitions, and then uses these guarded transitions when analyzing procedures that call them. The PREfix tool also performs symbolic execution of procedure bodies in order to extract the procedure summaries, but unlike dynamic symbolic execution tools, the bottom-up nature of the analysis and the tradeoffs for scalability

also means that the potential bug trace found by static analysis may not need to correspond to a reachable state in the program, i.e., there can be false warnings.

1.5.2 An Example

Let us illustrate the generation of bottom-up procedure summaries using Program 1.5.1. The procedure takes as input an unsigned integer `n` and returns a character array and a Boolean status. It assumes that `n` is at most 65535. This is ensured by the second `if` statement. It avoids memory allocation when `n` is 0, and returns `false` on allocation failure. The `InitName` procedure is called by `GetName`, whose purpose is to allocate the memory required for the `name` buffer and then copy over the contents from `source` into the newly allocated buffer. We will now see how static analysis tools can find potential problems with this code.

Program 1.5.1 Procedures InitName and GetName.

```
bool InitName(
    unsigned int n,
    char[]* outname)
{
  if (n == 0)
     return true;
  if (n > 65535)
     exit(1);
  *outname =
     alloc_array(char, n);
  if (*outname == 0)
     return false;
  return true;
}
```

```
char[] GetName(char* source,
               unsigned int n)
{
  bool success;
  char[] name;
  success = InitName(n, &name);
  if (!success) {
     return null;
  }
  strcpy(source, name);
  return name;
}
```

1.5.3 Methodolgy

There are two main components to the methodology. The first component consists of a bottom-up summarization of procedures. We illustrate how `InitName` is summarized as a set of guarded transitions. The second component is checking path feasibility. We illustate a path feasibility check on `GetName`. It uses the summaries from `InitName`.

```
assume (n == 0);
result = true;

assume (n != 0);
assume !(n > 65535);
assert (outname != null);
*outname =
    alloc_array(char, n);
assume (*outname == 0);
result = false;

assume (n != 0);
assume !(n > 65535);
assert (outname != null);
*outname =
    alloc_array(char, n);
assume (*outname != 0);
result = true;
```

```
outcome InitName_0:
  guards: n == 0
  results: result == true

outcome InitName_1:
  guards: n > 0 ∧ n <= 65535
  constraints: outname != null
  results: result == false

outcome InitName_2:
  guards: n > 0 ∧ n <= 65535
  constraints: outname != null
  results: result == true;
      is_alloc_array(char, n, *outname)
```

Fig. 1.1. Procedure summaries for InitName. Each block on the left corresponds to a path.

Summaries

Figure 1.1 covers the symbolic execution paths of `InitName` on the left and the corresponding guarded transitions on the right. The guarded transitions on the right represent the symbolic paths. Branch conditions that are recorded as assumptions along the path are represented as *guards*. The assertions, that come from run-time checks, are represented as *constraints*. Finally, the effect and return value of `InitName` is recorded as *results*.

For example, the first path represents the statement

```
if (n == 0) return true;.
```

The second path represents the case where allocation fails. It asserts that `outname` is a valid pointer, such that the dereferencing `*outname` is safe. The guards represent different execution paths and are therefore mutually exclusive. In the last two transitions, the guards don't imply the constraints `outname != null`, so it is possible to produce inputs that makes `InitName` encounter a run-time error. It depends on the context where `InitName` is used whether this is the case. Instead of flagging a warning outright, an *inter-procedural* static analysis method can check the constraints in context of the calling procedures, including `GetName`. This particular call site ensures that the pointer is valid.

Note that the symbolic execution path that calls `exit` does not correspond to a call that returns from `InitName`, so it is not included.

Checking Path Feasibility

The problem of checking whether a path is feasiable can be automatically reduced to checking the satisfiability of formulas derived from the source code, with the help of an SMT solver. We illustrate this process by checking path feasibility for the case of `GetName`.

The `strcpy` function requires that the `source` denotes a valid location in memory that has been initialized with a 0-terminated string. Similarly, `name` contains a valid memory allocation. So callers of `strcpy` must at the very least satisfy:

$$\texttt{init}(source) \wedge \texttt{is\_alloc\_array}(name) \ .$$

Yet, the following path through `GetName` is feasiable because the transition relation corresponding to `InitName_0` does not initialize `name` when $n = 0$. Feasibility can be checked by examining the symbolic path in Figure 1.2 (left). The extracted formula is on the right side. In the formula we have summarized the guarded transitions relations and taken their disjunction. Each transition relation is given as a conjunction of the guard and effect. The local `result` variable is returned, so we include an equality `success` = `result`.

```
success = InitName(&name, n);
assume success;
assume !init(name);
```

$$\left(\begin{array}{l} n = 0 \wedge \texttt{result} \\ \vee\ n > 0 \wedge n \leq 65535 \wedge \neg\texttt{result} \\ \vee\ n > 0 \wedge n \leq 65535 \wedge \texttt{result} \wedge \\ \quad is\_alloc\_array(char, n, name) \end{array}\right) \wedge$$
$$\texttt{success} = \texttt{result} \wedge$$
$$\texttt{success} \wedge$$
$$\neg\texttt{init}(name)$$

Fig. 1.2. A feasible path through GetName with a contract violation.

1.5.4 Interfacing with SMT Solvers

Our application to static analysis uses similar features as dynamic symbolic execution. The example also illustrated that the summary for `InitName` could be formulated as a disjunction, which is natively handled by SMT solvers.

1.5.5 Industrial Adoption

We presented static analysis from the point of view of Microsoft's PREfix tool [3]. It is used to analyze millions of lines of Microsoft source code on a routine basis. PREfix relies on a constraint solver to check for path feasibility. In this context Yannick Moy integrated Z3 as a bit-precise constraint solver. There are many other static analyzers, but we will only mention a few here.

The Coverity [11] analyzer contains analogous techniques as PREfix, including bit-precise analysis, as does GrammaTech's CodeSonar tool [3]. A common trait with these tools is that they don't aim to give strong guarantees about absence of runtime errors. They are bug-hunting tools. This contrasts with another class of static analysis methods based on abstract interpretation that have the ability to give strong guarantees for absence of a class of runtime errors. These tools rely on representing and propagating sets of reachable states as logical formulas or using specialized representations. The ASTREÉ [6] tool checks properties of numerical computations for floating point numbers. AdaCore's tool CodePeer [4] is also sound as it exhaustively checks for absence of runtime errors. In addition it detects some logical errors including dead code tests that always evaluate to true/false, and so on.

1.6 Program Verification

1.6.1 The Application

The symbolic exeuction, bounded model-checking, static analysis and software model checking methods we have examined so far have been relying on capabilities of automated search algorithms. They are automatic and scalable, but they handle typically only a class of specific properties or they may produce inconclusive results. Methods developed for program verification stand in contrast as they provide frameworks for establishing general logical specifications of programs. The downside is that one needs to find additional intermediary assertions in order to establish inductive invariants. The benefit is of course that non-trivial functional correctness can be established using program verification.

1.6.2 An Example

Consider the binary search program in Program 1.6.1.

[3] `http://www.grammatech.com/products/codesonar`
[4] `http://www.adacore.com/home/products/codepeer`

Program 1.6.1 A Binary Search program.

```
1  int BinarySearch(int[] a, int len, int key)
2  {
3    int low = 0;
4    int high = len;
5    while (low < high)
6    {
7      // Find middle value
8      int mid = low + (high - low) / 2;
9      int val = a[mid];
10     if (key < val) {
11         low = mid + 1;
12     } else if (val < key) {
13         high = mid;
14     } else {
15         return mid;
16     }
17   }
18   return -1;
19 }
```

We wish to establish partial correctness of the procedure. In other words, we should like to show that the procedure finds the index where the key resides. Partial correctness means that we here do not address the question whether the procedure terminates; it does but we will not be proving it here. Checking partial correctness means that given a set of pre-conditions that capture assumptions for the parameters, the program is free of runtime errors (such as illegal array access) and whenever the procedure terminates then either the `key` does not belong to the array and the procedure returns -1, or the `key` does belong to `a` and the procedure returns an index `result` within the bounds of the array such that `a[result] = key`. The assumptions are that the array `a` is sorted, and the interval from `low` until `high` is within the bounds of `a`. This contract can be formalized as *pre*- and *post*-conditions.

$$pre: \begin{array}{l} 0 \leq len \leq |a| \wedge \\ \forall i \in [0..len-2], j \in [i+1..len-1] \;.\; a[i] \leq a[j] \end{array} \tag{1.4}$$

$$post: \begin{array}{l} (0 \leq \texttt{result} \implies a[\texttt{result}] = \texttt{key}) \wedge \\ (\texttt{result} < 0 \implies \forall i \in [0..len-1] \;.\; a[i] \neq \texttt{key}) \end{array} \tag{1.5}$$

The program contains a loop and we will apply techniques that apply to general purpose loops without unrolling them into straight-line code. To establish the post-condition we will therefore need a *loop invariant* that gets

established the first time the loop is entered and is sufficiently strong to imply the post-condition. There are many techniques to infer loop invariants, including predicate abstraction and interpolation that were discussed in Section 1.4. Many of these techniques use SMT solvers in essential ways, but a description is beyond the scope of this chapter. We will here be content with using a loop invariant that our verification later establishes is sufficient.

$$\begin{array}{rl} & 0 \leq \mathit{low} \leq \mathit{high} \leq \mathit{len} \leq |a| \\ \mathit{inv}: & \wedge\, \forall i \in [0..\mathit{low}-1]\ .\ a[i] < \texttt{key} \\ & \wedge\, \forall i \in [\mathit{high}..\mathit{len}-1]\ .\ a[i] > \texttt{key} \end{array} \tag{1.6}$$

1.6.3 Methodolgy

We here describe an approach of veriyfing pre, post-conditions and loop invariants using predicate transformers. This approach converts an imperative program annotated with logical formulas into a logical formula. Note that there are other styles, such as proof rules in the style of Hoare [16], that produce several logical formulas from a given program.

We reduce a program annotated with assertions, pre, post-conditions and loop invariants into a core of program statements. We then use a weakest pre-condition calculus over these statements. The core consists of statements of the form

$$S, T ::= x = t \mid \texttt{havoc}\ x \mid \texttt{assert}\ \varphi \mid \texttt{assume}\ \varphi \mid S; T \mid S\ []\ T\ .$$

Notice that there are no while loops, no if-then-else conditions or return statements. On the other hand there is a new (non-deterministic) *choice* operator []. The operational meaning of $S\ []\ T$ is that computation is allowed to proceed non-deterministically either with S or T. We will later show how to reduce while loops and other constructs into this core. The core admits a straightforward logical semantics using the weakest liberal pre-condition predicate transformer *wp*.

$$\begin{array}{ll} \mathit{wp}(x = t, \varphi) & = \varphi[x \mapsto t] \\ \mathit{wp}(\texttt{havoc}\ x, \varphi) & = \forall x\ .\ \varphi \\ \mathit{wp}(\texttt{assert}\ \psi, \varphi) & = \psi \wedge \varphi \\ \mathit{wp}(\texttt{assume}\ \psi, \varphi) & = \psi \implies \varphi \\ \mathit{wp}(S; T, \varphi) & = \mathit{wp}(S, \mathit{wp}(T, \varphi)) \\ \mathit{wp}(S\ []\ T, \varphi) & = \mathit{wp}(S, \varphi) \wedge \mathit{wp}(T, \varphi) \end{array}$$

Programs like the binary search program in Program 1.6.1 can be translated into this core using the transformations that eliminate conditional statements and while-loops. The conditional statement

```
if (cond) S; else T;
```

is transformed into

```
{ assume cond; S; } [] { assume !cond; T; }
```

and we can check that it preserves the logical semantics.

We translate while loops annotated with invariants of the form

```
while (cond) inv S[x,y];
```

where, for illustrative purposes, we assume that S only modifies the two program variables x and y. The result is a loop-free program.

```
assert inv;          // check inv at loop entry
havoc x, y;          // fast forward to an arbitrary
assume inv;          // iteration of the loop
{
   assume guard;
   S;
   assert inv;       // check that inv is maintained
   assume false;
 []
   assume !guard;    // exit the loop
}
```

Return statements can be converted into assertions about the post-condition, such that the procedure body

```
τ2 p(τ1 x) { S; return t; T; }
```

becomes

```
τ2 p(τ1 x) { S; post[result := t]; assume false; T; }
```

Notice how using the statement **assume false**; allows the transformed program to ignore control flow after the return statement.

Program 1.6.2 shows the BinarySearch program transformed into the core language.

The result of applying the weakest pre-condition transformer is a pure formula. Let us show the weakest pre-condition transformer in action and extract it:

Program 1.6.2 Core representation of the Binary Search program.

```
int BinarySearch_C(int[] arr, int len, int key)
{
  l1: assume pre
  l2: int low = 0;
  l3: int high = len;
  l4: assert inv;
  l5: havoc low, high;
  l6: assume inv;
  l7:
  {
    m1: assume (low < high);
    // Find middle value
    int mid = low + (high - low) / 2;
    int val = a[mid];
    m2: {
      n1: assume (key < val);
      n2: low = mid + 1;
    []
      n3: assume !(key < val);
      n4: assume (val < key);
      n5: high = mid;
    []
      n6: assume !(key < val);
      n7: assume !(val < key);
      n8: assert post[result := mid]
      n9: assume false
    }
    m3: assert inv;
    m4: assume false;
  []
    m5: assume !(low < high);
  }
  l8: assert post[result := -1];
}
```

$$
\begin{array}{ll}
 & wp(\texttt{BinarySearch}_C(\texttt{a}, len, \texttt{key}), \textbf{true}) \\
\equiv & \{\text{expand } \texttt{BinarySearch}_C\} \\
 & wp(\ell_1;\ell_2;\ell_3;\ell_4;\ell_5;\ell_6;\ell_7;\ell_8, \textbf{true}) \\
\equiv & \{\ell_1 : \texttt{assume } pre\} \\
 & pre \implies wp(\ell_2\ell_3;\ell_4;\ell_5;\ell_6;\ell_7;\ell_8, \textbf{true}) \\
\equiv & \{\ell_2 : \texttt{int } low = 0;\} \\
 & pre \implies \textbf{let } low = 0 \ wp(\ell_3;\ell_4;\ell_5;\ell_6;\ell_7;\ell_8, \textbf{true}) \\
\equiv & \{\ell_3 : \texttt{int } high = len;\} \\
 & pre \implies \textbf{let } low = 0, high = len \ wp(\ell_4;\ell_5;\ell_6;\ell_7;\ell_8, \textbf{true}) \\
\equiv & \{\ell_4 : \texttt{assert } inv\} \\
 & pre \implies \textbf{let } low = 0, high = len \ inv \wedge wp(\ell_5;\ell_6;\ell_7;\ell_8, \textbf{true}) \\
\equiv & \{\ell_5 : \texttt{havoc } low, high\} \\
 & pre \implies \textbf{let } low = 0, high = len \ inv \wedge \forall low, high \ . \ wp(\ell_6;\ell_7;\ell_8, \textbf{true}) \\
\equiv & \{\ell_6 : \texttt{assume } inv\} \\
 & pre \implies \textbf{let } low = 0, high = len \ inv \wedge \forall low, high \ . \ inv \implies wp(\ell_7;\ell_8, \textbf{true}) \\
\equiv & \{\ell_8 : \texttt{assert } post[result \leftarrow -1]\} \\
 & pre \implies \textbf{let } low = 0, high = len \ inv \wedge \forall low, high \ . \ inv \implies wp(\ell_7, post[result \leftarrow -1])
\end{array}
$$

Continuing with the inner-most formula:

$$
\begin{array}{l}
\quad wp(\ell_7, post[result \leftarrow -1]) \\
\equiv \{\ell_7 : \{m_1; m_2; m_3; m_4\}\ []\ m_5\} \\
\quad wp(\{m_1; m_2; m_3; m_4\}\ []\ m_5, post[result \leftarrow -1]) \\
\equiv \{\text{by semantics of } []\} \\
\quad wp(m_1; m_2; m_3; m_4, post[result \leftarrow -1]) \wedge wp(m_5, post[result \leftarrow -1]) \\
\equiv \{m_5 : \texttt{assume}\ \neg(low < high)\} \\
\quad wp(m_1; m_2; m_3; m_4, post[result \leftarrow -1]) \wedge (\neg(low < high) \implies post[result \leftarrow -1]) \\
\equiv \{m_4 : \texttt{assume}\ \textbf{false}\} \\
\quad wp(m_1; m_2; m_3, \textbf{true}) \wedge (\neg(low < high) \implies post[result \leftarrow -1]) \\
\equiv \{m_3 : \texttt{assert}\ inv\} \\
\quad wp(m_1; m_2, inv) \wedge (\neg(low < high) \implies post[result \leftarrow -1]) \\
\equiv \{m_1 : \texttt{assume}\ low < high\} \\
\quad (low < high \implies wp(m_2, inv)) \wedge (\neg(low < high) \implies post[result \leftarrow -1]) \\
\equiv \{\text{by a full unfolding of } wp(m_2, inv)\} \\
\quad \left(\begin{array}{l}
low < high \implies \\
\quad \textbf{let}\ mid = low + (high - low)/2 \\
\quad \textbf{let}\ val = \texttt{a}[mid] \\
\quad\quad (\texttt{key} < val \implies inv[low \mapsto mid + 1]) \wedge \\
\quad\quad (\neg(\texttt{key} < val) \wedge (val < \texttt{key}) \implies inv[high \mapsto mid]) \wedge \\
\quad\quad (\neg(\texttt{key} < val) \wedge \neg(val < \texttt{key}) \implies post[result \mapsto mid])
\end{array}\right) \wedge \\
\quad (\neg(low < high) \implies post[result \leftarrow -1])
\end{array}
$$

We have now enough information to extract the full formula corresponding to partial correctness of the program:

$$
\begin{array}{l}
pre \implies \\
\textbf{let}\ low = 0 \\
\textbf{let}\ high = len \\
inv \wedge \\
\left(\begin{array}{l}
\forall low, high\ .\ inv \implies \\
\quad \neg(low < high) \implies post[result \mapsto -1] \\
\quad \wedge \left(\begin{array}{l}
low < high \implies \\
\quad \textbf{let}\ mid = low + (high - low)/2 \\
\quad \textbf{let}\ val = \texttt{a}[mid] \\
\quad\quad (\texttt{key} < val \implies inv[low \mapsto mid + 1]) \wedge \\
\quad\quad (\neg(\texttt{key} < val) \wedge (val < \texttt{key}) \implies inv[high \mapsto mid]) \wedge \\
\quad\quad (\neg(\texttt{key} < val) \wedge \neg(val < \texttt{key}) \implies post[result \mapsto mid])
\end{array}\right)
\end{array}\right)
\end{array}
$$

When we plug in the definitions for *pre*, *post* and *inv* from (1.4), (1.5) and (1.6) we obtain a valid first-order formula.

1.6.4 Bit-precise reasoning

Bit-vectors can be used to accurately encode the semantics of machine integer operations. SMT solvers support directly a theory of bit-vectors that

corresponds to machine integer semantics. As an illustration of the importance of bit-precise analysis, consider line 8 in Program 1.6.1. It contains the conspicuous assignment `int mid = low + (high - low) / 2;`. If `low` and `high` were genuine integers, then the usual arithmetical laws would tell us that the assignment is equivalent to `int mid = (low + high)/2;`. However, the two assignments are not interchangeable when `int` ranges over 32-bit integers even when we know that `low <= high`. Namely, take the value 2^{30} for both `high` and `low`, then in two-complements arithmetic `high+low` evaluates to -2^{31} (the most significant bit is 1, which produces the negative sign), and therefore `(high+low)/2` is -2^{30}. On the other hand `low + (high - low)/2` evaluates to 2^{30}, which is what we would like.

1.6.5 Industrial Adoption

The ideal of verified programs is a long-running quest since the advent of program verification [17]. While the ideals are especially pursued in a scientific context, there has also been a steady pursuit in the context of of civil and defense industry. Some of the most significant recent advances have been in verification of compilers and operating systems code. The CompCert project [20] has pionered formal verification of compilers using the interactive Coq system based on the calculus of constructions. The LLVM operating system kernel [19] was also verified using the Isabelle theorem prover for higher-order logic. From the perspective of automation, SMT solvers have found their way from extended type checking style verification from the Extended Static Checker (ESC) tool using the Simplify SMT solver [8] towards verification of mainstream programming languages C# and C using the tool Boogie [7] that maps low level imperative programs to first-order logic in the form we demonstrated and the verifying C compiler (VCC) [10] and HAVOC [5] that convert C programs into the Boogie intermediary language format. The VCC tool was used to verify functional correctness of large portions of Microsoft's Viridian Hyper-V. The SPARK-ADA tools similarly map program verification tasks for ADA programs into first-order verification conditions that are solved using a variety of theorem provers, and Frama-C [2] uses the Why intermediary format for bridging verification conditions from C into first-order logic formulas. SMT solvers have thanks to their integration of first-order with theory reasoning over domains used in programming languages been instrumental for program verification. The quest for verified software goes on and the area of verifying termination, asymptotic runtime characteristics, heap manipulating and multi-threaded programs is an active area of research.

1.7 Bibliographical notes

Floyd's approach associated a proposition with each connection in a control flow graph. Hoare logic uses triples of a program statement and pre and post

conditions. Dijkstra [9] introduced the perspective that statements in imperative programs correspond to *predicate transformers*. These are used to map a post-condition assertion to a weakest pre-condition formula that has to be satisfied in order for the post-condition to be entailed. King [18] introduced symbolic execution in the context of program testing. Manna and Pnueli [21] developed methods for establishing temporal properties of reactive and concurrent programs.

Our use of a substrate of the C, C++, C# and Java languages was inspired by the C0 language by Frank Pfenning. It is used for teaching imperative programming for freshmen at CMU[5]. Rigorous verification of properties of C programs requires an equally rigorous formalization of the relevant semantics. In the context of verified compilers into C[6]. The VCC [10] system formalizes and implements a significant portion of the C semantics.

[5] http://www.cs.cmu.edu/~fp/courses/15122-f10/misc/c0-reference.pdf

[6] http://gallium.inria.fr/~xleroy/publi/Clight.pdf

References

1. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, 2001. SLAM.
2. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/download/acsl_1.4.pdf`.
3. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.
4. C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *OSDI*, pages 209–224. USENIX Association, 2008.
5. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In Z. Shao and B. C. Pierce, editors, *POPL*, pages 302–314. ACM, 2009.
6. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *European Symposium on Programming (ESOP)*, 2005.
7. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report 2005-70, Microsoft Research, 2005.
8. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
9. E. W. Dijkstra. *A discipline of programming.* Prentice Hall, 1976.
10. 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.
11. D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.
12. R. Floyd. Assigning meanings to programs. *Proceedings of the Symposia in Applied Mathematics*, 19:19–32, 1967.
13. 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.

14. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254, June 1997.
15. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002. BLAST.
16. C. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.
17. T. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, January 2003.
18. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
19. G. Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, Feb. 2009.
20. X. Leroy. The compcert project.
21. Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, London, 1981.
22. A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The Yogi Project: Software Property Checking via Static Analysis and Testing. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 178–181. Springer, 2009.