Checking Temporal Properties of Software with Boolean Programs

Thomas Ball Sriram K. Rajamani
{tball,sriram}@microsoft.com

Software Productivity Tools
Microsoft Research

Abstract. A fundamental issue in model checking of software is the choice of a model for software. We present a model called *boolean programs* that is expressive enough to capture interesting properties of programs and is amenable to model checking. We present a model checking algorithm for boolean programs using context-free-language reachability. The model checking algorithm allows procedure calls with unbounded recursion, exploits locality of variable scopes, and gives short error traces. Furthermore, we give a process for incrementally refining an initial skeletal boolean program B (representing a source program P) with respect to a particular reachability query in P. The presence of infeasible paths in P may lead to the model checker reporting false positive errors in B. We show how to refine B by introducing boolean variables to rule out the infeasible paths. The process uses ideas from model checking and symbolic execution to automatically perform predicate abstraction.

1 Introduction

What would model checking of software look like if it were a well established practice today? If it had the attributes that model checking of hardware circuits does, it might look like this:

- There would be a representation **R** for modeling software, analogous to the finite state machines (FSMs) used for modeling hardware circuits, and efficient algorithms to model check **R**.
- The model checking algorithms over **R** would report a shortest trace to an error when they find errors, as model checkers for FSMs do.
- Programming languages such as C, C++ and Java would have translations into **R**, just as hardware description languages such as VHDL and Verilog can be compiled into FSMs.
- An instance r in **R** could be refined into an instance r' in **R** and proved correct (either by construction, or by verification), just as FSMs can be refined and proved correct using the semantics of trace inclusion.
- The model checking algorithms on **R** would be able to exploit the inherent modularity and abstraction boundaries present in the source programs for efficiency.

We investigate how to model check temporal properties of sequential programs with an eye on these five desiderata. We consider programs written in imperative languages such as C and Java. In particular, we are interested in analyzing programs containing recursion, pointers (references) and dynamic memory allocation. We define a target representation **R** called *boolean programs*. Informally, boolean programs are C programs in which all variables and parameters (call-by-value) have boolean type. In addition, boolean programs have a form of control

```
     void useUnit(int *units) {
       int canEnter = 0;
L1:    if (*units == 0) {
L2:      canEnter = bump(units);
       } else {
L3:      canEnter = 1;
       }
L4:    if (canEnter) {
L5:      if (*units == 0) {
ERR:       assert(0);
         } else {
L6:        consumeUnit();
         }
       }
     }

     int bump(int *u) {
       int level = getLevel();
M1:    if (level > 0) {
M2:      getUnit();
M3:      *u = *u + 1;
M4:      return(1);
       } else
M5:      return(0);
     }
```

P

```
     void useUnit() begin

L1:    if (*) then
         skip;
L2:      bump();
       else
         skip;
L3:      skip;
       fi
L4:    if (*) then
         skip;
L5:      if (*) then
           skip;
ERR:       assert(0);
         else
           skip;
L6:        skip;
         fi
       fi
     end

     void bump() begin
M1:    if (*) then
M2:      skip;
M3:      skip;
M4:      return;
       else
M5:      return;
       fi
     end
```

B_1

```
     decl {*M=0};
     void useUnit() begin
       decl {u=M} := 1;
L1:    if (*) then
         assert(!({u=M} & !{*M=0}));
L2:      bump();
       else
         assert(!({u=M} & {*M=0}));
L3:      skip;
       fi
L4:    if (*) then
         skip;
L5:      if (*) then
           assert(!({u=M} & !{*M=0}));
ERR:       assert(0);
         else
           assert(!({u=M} & {*M=0}));
L6:        skip;
         fi
       fi
     end

     void bump() begin
M1:    if (*) then
M2:      skip;
M3:      {*M=0} := unk();
M4:      return;
       else
M5:      return;
       fi
     end
```

B_2

Fig. 1. An example program P and three boolean programs (B_1, B_2 and B_3) that abstract P with an increasing level of precision.

non-determinism. What distinguishes boolean programs from FSMs is that boolean programs contain procedures with recursion. As procedural abstraction and recursion are key components of all modern programming languages, we believe that boolean programs provide a good starting point for investigating model checking of software.

Our goal is to check whether a sequential program P obeys a temporal property Φ. It is well-known that this problem can be reduced to the problem of invariant checking (assertion violation) in an instrumented version of P (a version of P containing extra code that simulates the FSM corresponding to the complement of the property Φ). Therefore, we will focus on the problem of checking whether or not a statement (which might correspond to the accept state of the FSM) is reachable in P.

To make our goal concrete, we consider the C program P in Figure 1. We are interested in knowing if the statement "`assert(0)`" at label `ERR` is reachable, as this would indicate

```
   decl {*M=0};
   void useUnit() begin
     decl {u=M} := 1,{cE!=0} := 0;
L1:  if (*) then
       assert(!({u=M} & !{*M=0}));
L2:    {cE!=0} := bump();
     else
       assert(!({u=M} & {*M=0}));
L3:    {cE!=0} := 1;
     fi
L4:  if (*) then
       assert({cE!=0});
L5:    if (*) then
         assert(!({u=M} & !{*M=0}));
ERR:     assert(0);
       else
         assert(!({u=M} & {*M=0}));
L6:      skip;
       fi
     fi
   end

   bool bump() begin
M1:  if (*) then
M2:    skip;
M3:    {*M=0} := unk();
M4:    return (1);
     else
M5:    return (0);
     fi
   end
```

B_3

```
   decl {*M=0};
   void useUnit() begin
     decl {u=M} := 1,{cE!=0} := 0;
L1:  if (*) then
       assert(!({u=M} & !{*M=0}));
L2:    {cE!=0} := bump({u=M});
     else
       assert(!({u=M} & {*M=0}));
L3:    {cE!=0} := 1;
     fi
L4:  if (*) then
       assert({cE!=0});
L5:    if (*) then
         assert(!({u=M} & !{*M=0}));
ERR:     assert(0);
       else
         assert(!({u=M} & {*M=0}));
L6:      skip;
       fi
     fi
   end

   void bump({u=M}) begin
M1:  if (*) then
M2:    skip;
M3:    {*M=0} := H(!{u=M}&{*M=0},
         (!{u=M}&!{*M=0})|({u=M}&{*M=0}));
M4:    return (1);
     else
M5:    return (0);
     fi
   end
```

B_4

Fig. 2. The last two boolean programs B_3 and B_4.

that a "Unit" was acquired improperly. (The label `ERR` is not reachable since the invariant $(\texttt{canEnter!} = 0) \Rightarrow (*\texttt{units!} = 0)$ holds in the program; we will discover this invariant through a process that can be automated.) We start with a boolean program that coarsely abstracts P and incrementally refine it, driven by the goal of answering our reachability query:

– We first generate the "skeletal" boolean program B_1 from P.[1] Program B_1 retains the control-flow structure of P. However, every variable declaration of P has been removed, every assignment statement has been replaced by **skip** and all predicate expressions in conditional statements have been replaced by "*" (non-deterministic boolean choice). In addition, **skip** statements have been introduced immediately after conditions (for reasons

[1] Boolean programs are written using a Pascal-like syntax to distinguish them from the C programs from which they are derived. In boolean programs, 1 denotes "true" and 0 denotes "false".

that will become clear soon). We say that B_1 abstracts P as the set of feasible (executable) paths in B_1 is a superset of the set of feasible paths in P.

- We now ask the question: is the label `ERR` reachable in B_1? A model checker over B_1 might be used to answer this query. The answer is "yes" and one shortest path leading to `ERR` is `[L1,L3,L4,L5,ERR]`. However, this path is infeasible in P because it constrains the value `*units` to be both equal to `0` (by the control transition from label `L5` to `ERR`) and not equal to `0` (label `L1` to `L3`).
- We want to refine program B_1 to eliminate this infeasible path, while ensuring that all feasible paths in P are feasible in the new boolean program. What program state should we choose to model in the boolean program? The condition `(*units==0)` is an obvious choice. The variable `units` is a pointer, so the condition really is about the state of the memory pointed to by `units`. Therefore, we create two boolean variables: a global variable {`*M=0`} models the fact that the value in some memory location M is `0`; a variable {`u=M`} (local to `useUnit`) models the fact that the value of the variable `units` is M.[2] Given these variables, program B_2 is constructed from P by determining how each statement of P affects the values of the conditions represented by the boolean variables.
- In program B_2, the path `[L1,L3,L4,L5,ERR]` is infeasible. Why? First, note that the boolean variable {`u=M`} is always true, as it is never modified (once initialized) in B_2. The transition from label `L1` to `L3` implies `!(`{`u=M`} & {`*M=0`}`)` (due to the assert statement immediately before label `L3`; the purpose of the assert statements insert in the arms of conditional statements is to prune away infeasible paths). Since {`u=M`} $= 1$, this implies that {`*M=0`} $= 0$. This rules out the transition from label `L5` to `ERR`, which requires that `!(`{`u=M`} & !{`*M=0`}`)`.

  Of special interest is the translation of the assignment statement `*u = *u+1` at label `M3` in program P. An alias analysis of program P reveals that the formal parameters `units` of procedure `useUnit` and `u` of procedure `bump` are aliases. Thus, the assignment at `M3` can affect the value in memory location `M`, which affects the truth value of the global variable {`*M=0`}. However, the way in which the assignment affects its value is (currently) unknown. This is modeled by assigning to the variable {`*M=0`} a non-deterministic value returned by the following function `unk`:

```
bool unk() begin
  if (*) then return 0; else return 1; fi
end
```

- We again ask the question: is `ERR` reachable in B_2? The answer is still "yes" due to the path `[L1,L2,M1,M5,L4,L5,ERR]`. This path is infeasible in P as the variable `canEnter` remains `0`, while the control transition from label `L4` to `L5` is taken, indicating that `(canEnter!=0)`, a contradiction. This suggests that we need to add a boolean variable to model this condition, namely {`cE!=0`}.
- The boolean program B_3 of Figure 2 is the result of transforming the source program to correctly update variables {`cE!=0`}, {`*M=0`} and {`u=M`}. Note that the return value of the procedure `bump` is now modeled, whereas it was ignored in B_2. The label `ERR` still is reachable

[2] In boolean programs, $\{s\}$ is a variable identifier, where s can be an arbitrary string not containing }, { or whitespace.

in B_3 by the path [L1,L2,M1,M2,M3,M4,L4,L5,ERR]. In this path in P, *units==0 on entry to bump and *units!=0 on exit from bump. Thus, the transition from L5 to ERR is not possible and the path is infeasible.

– Program B_4 is obtained from B_3 by adding the parameter {u=M} to bump. Label ERR is not reachable in program B_4, which implies that ERR is not reachable in P (as B_4 is an abstraction of P). This program has few changes when compared to B_3: only the statement at label L2 is changed in the first procedure (in order to pass the value of the local variable {u=M} to the procedure bump). Program B_4 correctly models the fact that execution of the assignment statement at M3 ensures that {*M=0} is 0 on exit from bump. Let us explain the function H at label M3 in B_4, which assigns to the variable {*M=0}. The code for function H is given below:

```
bool H(e,f) begin
  if    (e) then return 1;
  elsif (f) then return 0;
  else           return unk();
  fi
end
```

The only way in which {*M=0} is 1 after the assignment statement *u = *u+1 is if the value in memory location M is 0 before the assignment and variable u does not point to M. This is modeled by the first parameter to the H function: !{u=M} & {*M=0}. The second parameter to the H function models the situations in which the value of the memory location M is guaranteed to be not equal to 0. When neither of these parameters is 1, the function returns 0 or 1 nondeterministically.

In the above example, we have sketched a process for refining a program P into a boolean program B based on the goal of invariant detection ("label ERR is not reachable in P"). The process starts with a skeletal boolean program B with no boolean variables. The boolean program is incrementally refined using the following iterative procedure:

1. Is there a path p in B that witnesses a violation of the invariant? If not, then the invariant holds in P. This can be determined via model checking of the boolean program B.
2. If there is such a path p, is p is feasible in P? There are three possible outcomes to this undecidable question, which generally is answered using an automated heuristic decision procedure:
   - Path p is feasible in P. In this case, an error in P has been found and the process terminates.
   - Path p is infeasible in P. In this case, the process continues at step (3).
   - The decision procedure used to determine the feasibility/infeasibility returns the answer "don't know". In this case, the process terminates with a "don't know" answer.
3. What are the conditions that imply the infeasibility of path p in P?
4. Using these conditions, create boolean variables to model the conditions and transform the program P into a new boolean program B' such that p is infeasible in B', B abstracts B', and B' abstracts P.
5. Replace B with B' and go back to 1.

Due to the small size of program P in Figure 1, a model checker applied directly to program P (or an encoding of P in a language such as Promela) might determine that label `ERR` is not reachable in P. Of course, in general, it is undecidable to check if a label in a C/Java program is reachable. Even if P has only variables over finite domains then model checking over P may be impractical if the number of variables is large. In such situations, it is useful to construct abstractions of P, as our process does.

The results of this paper, in addition to this refinement process, are solutions to steps 1, 2 and 4. For step 3, we rely on existing techniques for symbolically executing a single path (such as in [CR81,DE82]) and determining its feasibility, referred to here as *path simulation*. Our results are the following:

- *Model checking of a boolean program with procedures and recursion.* We transform the reachability problem for boolean programs to context-free language(CFL) reachability [RHS95] and obtain a model checking algorithm for boolean programs. The model checking algorithm is cubic in the size of the control flow graph and exponential in the maximum number of *local* variables in the program. Further, it provides short error traces when errors are found. We have implemented this algorithm in `Bebop`, a model checker for boolean programs, which uses BDDs to perform CFL-reachability symbolically.
- *Identification of a set of conditions that imply the infeasibility of a path.* We show how a path simulator can be used to determine the feasibility/infeasibility of a path and how it can supply a set of conditions that "explains" the infeasibility of a path.
- *Derivation of a boolean program from the infeasibility conditions.* Given the information from the above step, we show how to construct a boolean program B from P (by local transformations using weakest preconditions) such that B abstracts P and in which a particular path p (that is infeasible in P) becomes infeasible. In general, several infeasible paths in P could become infeasible in B through this construction, especially if few boolean variables are needed to explain the infeasibility.

The remainder of this paper discusses how the Steps 1–4 in the above iterative procedure may be automated through a discussion of the above example. Section 2 discusses our model checking algorithm for boolean programs based on CFL-reachability (Step 1) and its implementation in the model checker `Bebop`. Section 3 describes the operation of a path simulator (Step 2). Section 4 explains how iterative refinement (Steps 3 and 4) is automated. Section 5 reports on the status of our implementation and the application of our process to the domain of device drivers. Section 6 discusses related work.

2 Model Checking of Boolean Programs

An instance $\langle B, l\rangle$ of the *boolean program reachability problem* consists of a boolean program B, and a label l in B. The answer to the problem is "yes" if l is reachable in B and "no" if l is not reachable in B. We give a decision procedure to answer the boolean-program-reachability problem that has the characteristics of model checking. In particular, if l is reachable, the procedure yields a trajectory that proves the reachability of l.

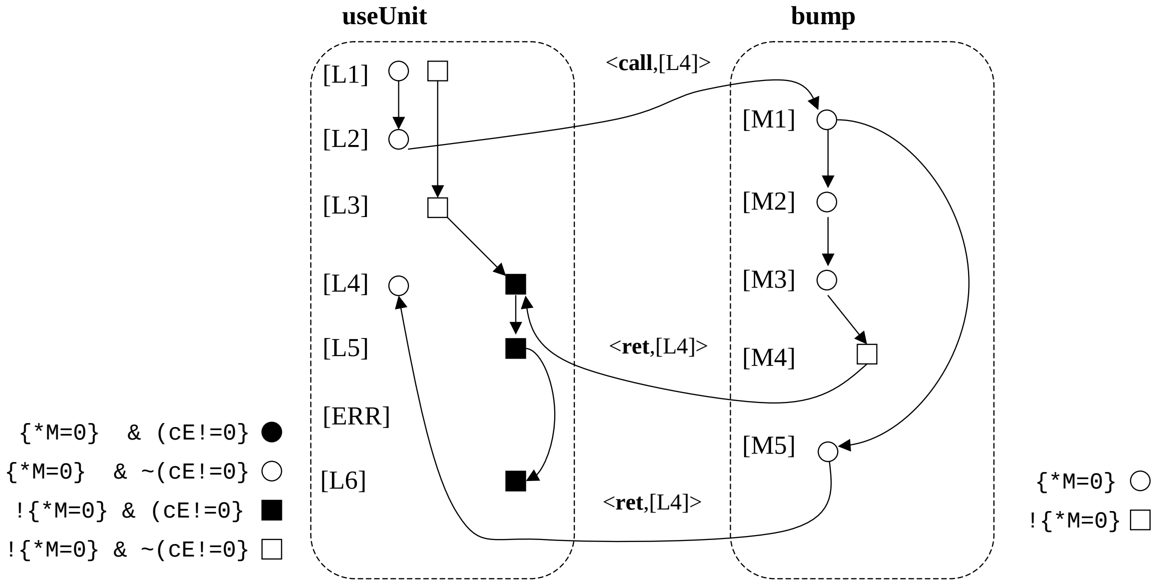

Fig. 3. Exploded graph of program B_4 from Figure 2, showing that statement label `ERR` is not reachable. White nodes denote states in which `{cE!=0}` is `0` while black nodes denote states in which `{cE!=0}` is `1`. A round node denotes states in which `{*M=0}` is `1` while a square node denotes states in which `{*M=0}` is `0`. The local variable `{u=M}` is not represented since it is always `1`. In procedure `bump` the color of the nodes is immaterial (`{cE!=0}` is not in scope), and `bump`'s formal parameter is always `1`.

Let $\langle B, l\rangle$ be an instance of the boolean program reachability problem. In our technical report, we show how $\langle B, l\rangle$ can be reduced to a CFL-reachability problem in a graph representing the state space of B [BR00]. Conceptually, CFL-reachability works over an *exploded graph* $G_B = \langle N_B, A_B\rangle$, where the nodes N_B are the states of the boolean program B and A_B are state transitions. Figure 3 shows the exploded graph representation of the boolean program B_4 from Figure 2. The key insight from CFL-reachability is that (recursive) procedures can be analyzed efficiently by computing "summaries" of how they transform program state. These summaries have two effects: (1) they allow procedures to be analyzed independently; (2) they allow analysis of programs containing recursion. If B has a constant number of global variables then the running time of our model checker is asymptotically exponential in the maximum number of local variables (over all procedures) in B. Thus, the model checking exploits the inherent modularity from procedural abstraction.

We have adapted the CFL-reachability algorithm of Reps et al. [RHS95], which works in a directed manner from the **main** procedure. We have implemented this algorithm symbolically using Binary Decision Diagrams(BDDs) [Bry86] in our model checker `Bebop`. `Bebop` uses a mixture of explicit and implicit state representations. The control flow graph is represented explicitly, and the set of all "path edges" [RHS95] that are incident on a given node of the control flow graph is represented implicitly as a BDD attached to the node. The implementation closely resembles inter-procedural data-flow analysis. However, the number of facts at each node is exponential in the number of variables in scope at that node, and BDDs are used to represent sets of these facts. While there are changes to the BDDs that represent the set of data-flow facts, a statement with a changed BDD is selected and its transfer function is used to update the

p	Env	$Cond$
`canEnter = 0;`	$\langle canEnter, 0\rangle$	
`L1: assert(!(*units == 0));`	$\langle units, M\rangle$	*M!=0
`L3: canEnter = 1;`	$\langle canEnter, 1\rangle$	
`L4: assert(canEnter);`		
`L5: assert(*units == 0);`		*M=0

Fig. 4. Symbolic execution of the trace `[L1,L3,L4,L5,ERR]` in program P from Figure 1.

BDDs representing the data-flow facts at the successor nodes. At procedure calls, if a compatible summary exists the procedure need not be reanalyzed. Otherwise, the procedure is analyzed and the summary information computed. We have modified the CFL-reachability algorithm to keep track of the length of the shortest path needed to reach each state, so that if l is reachable in B, the algorithm will give a shortest trajectory that ends in a state labeled l.

3 Path Simulation and Infeasible Paths

A given error trace γ in a boolean program B (that abstracts a source program P) may not be a trace of P. We use the technology of path simulation (symbolic evaluation [CR81,DE82] of a single path through P) to determine if γ is indeed a trace (feasible path) of P. Recall from Section 1 that if γ is a feasible path of P then an error has been found in P. If γ is not feasible then we use the path simulator to find a set of expressions E to construct a boolean program B' from P such that γ is not a trace of B'.

We represent an error trace through P by a sequence (path) of assignment statements and assert statements. Assignments statements in the path may represent assignment statements in P or the implicit assignment of actual parameters to formal parameters. Assert statements in the path may represent assert statements in P or conditional expressions in control flow statements. For example, if the condition $(x < y)$ of a **while** loop evaluates false in γ, this would be modeled by "`assert(!(x<y))`" in the path corresponding to γ.

We explain the operation of a path simulator through the example program P of Figure 1. Consider the trace `[L1,L3,L4,L5,ERR]`, which model checking of B_1 will return as a potential error trace. Figure 4 shows the path in P corresponding to this trace and the state that a path simulator maintains when analyzing the path (from top to bottom).

A path simulator maintains an environment (*Env*) mapping variables to symbolic values and a set of conditions (*Cond*) over these symbolic values. As shown in Figure 4, the first statement in the path is the assignment to the variable `canEnter`, resulting in an update to *Env*. Consider the next statement, which is the assert at label `L1`. Since the pointer `units` has not been initialized, the path simulator invents a symbolic value M for the variable and updates *Env*. Furthermore, as the assert constrains the contents of M to be non-zero, this constraint is added to *Cond*. The execution of the assignment statement at label `L3` updates the value of *Env* to reflect the new value of variable `canEnter`. The assert statement at label `L4` succeeds because the value of `canEnter` in the current environment is 1 (this assert adds no new information to the state of the path simulator). Finally, the assert at label `L5` will fail because: (1) the value of units is in *Env* is M; (2) the constraint *M!=0 already is present in *Cond*. As a result, the path is infeasible.

The infeasibility of a path can be "explained" as a conjunction of conditions E derived from the state of the path simulator that imply the falsehood of an assert expression. The details of how E is constructed are given in [BR00]. For our example, the condition $E = (units = M) \wedge !(*M = 0)$ implies that assert(*units==0) at label L5 will fail. In other words, $E \wedge (*units = 0)$ is not satisfiable. Note that the value of the variable canEnter does not affect the infeasibility of the path. In general, we wish to find a minimal set of conditions that "explains" a path's infeasibility (see [BR00] for details). Path simulators use a decision procedure to show that a conjunction of conditions cannot be satisfied. Satisfiability is undecidable in general, but there are heuristic decision procedures that work well in practice. A heuristic decision procedure may report three possibilities: "satisfiable", "not satisfiable", "don't know".

4 Iteratively Refining Boolean Programs

Let P and Q be programs with the same set of labels. We say that P *refines* Q, or Q *abstracts* P, written $P \preceq Q$, if every feasible path of P is a feasible path of Q.

In our example from Figure 2, it is obvious that $P \preceq B_1$. In the previous section, we showed how path simulation can be used to determine the feasibility of a path and, when a path is infeasible, to find a set of conditions E that explains the infeasibility. We now explain how to generate the program B_2 (of Figure 1) from the set of conditions $E = \{(units = M), (*M = 0)\}$ and the program P. Corresponding to the conditions in E we introduce two boolean variables {u=M} and {*M=0} in B_2. The goal of the refinement algorithm is to ensure that (1) path [L1,L3,L4,L5,ERR] is infeasible in B_2, (2)$P \preceq B_2$, and (3)$B_2 \preceq B_1$.

First, consider the translation of the conditional expressions in program P. Take, for example, the conditional expression (*units == 0) at label L1. Each conditional expression e in P gives rise to two assert statements in the boolean program, modeling the two possible outcomes of the expression.[3] In B_2, these two assert statements immediately precede labels L2 and L3. The purpose of these assert statements is to prune away infeasible paths. For example, if the condition (*units == 0) evaluates to true then it cannot be the case that $units = M$ and $*M! = 0$. Thus, the expression in the assert statement just before label L2 should be !({u = M}&!{*M = 0}). Dually, if the condition at label L1 evaluates to false then it cannot be the case that $units = M$ and $*M = 0$.

Second, consider the translation of the various assignment statements in P. Some assignments, such as those to the variable canEnter, cannot possibly affect the conditions in E. The condition at label M3 can affect the outcome of the condition $*M = 0$, since the variable u is an alias to the variable units in procedure useUnit. However, the knowledge encoded in the boolean variables in B_2 is not sufficient to determine either the truth or falsehood of this condition. Therefore, we assign 0 or 1 to {*M=0} nondeterministically via the function unk().

For a statement s and a boolean expression e, let $WP(s, e)$ denote the *weakest precondition* [Gri81] of e with respect to statement s. In general, given an assignment statement s an

[3] In general, the evaluation of a conditional may introduce new knowledge into the boolean program, thus affecting the values of boolean variables. This does not happen in our example. See [BR00] for details.

expression e (representing a set of program states), and a boolean variable b that models e, the translation of an assignment statement requires finding the weakest set of conditions F (as represented by expressions in E) such that F implies $WP(s,e)$. Consider the assignment `*u = *u+1` at label `M3`, and its translation in the boolean program B_4. In B_4, we have two boolean variables `{u=M}` and `{*M=0}` that are in scope at label `M3`. Thus, the set E of relevant expressions is $\{(u = M), (*M = 0)\}$. We are interested in modeling the effect of this assignment on the variable `{*M=0}`. We have that $WP(*\mathtt{u} = *\mathtt{u} + \mathtt{1}, (*M = 0))$ is given by $((u = M)\&(*M = -1)) \mid (!(u = M)\&(*M = 0))$. Further, $!(u = M)\&(*M = 0)$ is the weakest expression over E that implies $WP(*\mathtt{u} = *\mathtt{u} + \mathtt{1}, (*M = 0))$. Similarly, $(!(u = M)\&!(*M = 0)) \mid ((u = M)\&(*M = 0))$ is the weakest expression over E that implies $WP(*\mathtt{u} = *\mathtt{u} + \mathtt{1}, !(*M = 0))$. Thus, the assignment is translated to:

$$\{*\mathtt{M} = \mathtt{0}\} := \mathtt{H}(!\{\mathtt{u} = \mathtt{M}\}\&\{*\mathtt{M} = \mathtt{0}\}\ ,\ (!\{\mathtt{u} = \mathtt{M}\}\&!\{*\mathtt{M} = \mathtt{0}\}) \mid (\{\mathtt{u} = \mathtt{M}\}\&\{*\mathtt{M} = \mathtt{0}\}))$$

Since symbolic values are introduced by the path simulator, the set E of conditions can refer to symbolic values as well. We introduce *annotations* to relate program variables to symbolic values. For example, the declaration `decl {u=M}:= 1` preceding label `L1` in B_2 is one such annotation (for the symbolic value M denoting the memory location that the variable `units` points to). A detailed description of annotations can be found in [BR00].

5 Current Status

Our current objective is to use the methodology in this paper to show that device drivers for operating systems obey certain temporal properties. Device drivers form the interface between an operating system and the devices that the operating system administers and controls. Most device drivers are created by companies other than operating systems vendors. Because drivers run in kernel space, a misbehaving driver can wreak havoc. Although a device driver operates in a multi-threaded environment, the majority of properties it must obey concern its sequential behavior as it performs calls to operating system functions.

As mentioned in Section 2, we have designed and implemented `Bebop`, a model checker for boolean programs, to automate Step 1 of our process. We have a path simulator at Microsoft Research to do Step 2. We have been able to check a temporal property of a device driver for the serial port from the Windows NT DDK, by doing Steps 3 and 4 manually. We are currently working on automating these steps and performing more experimentation.

6 Related Work

Model checking for state machines is a well studied problem, and several model checkers — SMV [McM93], Murϕ [Dil96], SPIN [HP96], COSPAN [HHK96], and MOCHA [AHM+98]— have been developed. Several notions of refinement, including trace containment, have been studied before [Mil71] [AL88] [Kur94]. Boolean programs can be viewed as abstract interpretations of the underlying program [CC77]. The connections between model checking, dataflow analysis and abstract interpretation have been explored before [Sch98] [CC00]. The model checking problem for pushdown automata has been studied before [SB92] [BEM97] [FWW97].

The CFL reachability framework from [RHS95] builds on earlier work in interprocedural dataflow analysis from [KS92] and [SP81]. However, our use of CFL reachability as a model checking procedure for boolean programs is new. Exploiting design modularity in model checking has been recognized as a key to scalability of model checking [AH96] [AG00] [McM97] [HQR98]. The idea of using a boolean program to harness the inherent modularity in procedural abstraction, and exploiting locality of variable scoping for efficiency in model checking is new.

Predicate abstractions were introduced in [GS97]. We use predicates that refer to both program variables and symbolic variables introduced during path simulation. Further, annotations are used to relate symbolic variables to program variables. This simplifies the automatic construction of abstractions.

Program slicing is a projection operation on programs that preserves the execution behavior of statements in the projection [Wei82]. The semantics of slicing generally guarantees that the projection will have "identical" behavior to the original program, if started in an "identical" state. This form of slicing is used by Hatcliff and Dwyer [HD99] in their work on constructing models from programs. If we apply their algorithm to the example program P of Figure 1, slice would contain all the variables. In particular, the slice would determine that the variable *level* is relevant in determining the reachability status of label `ERR`, which is not true, as the abstraction B_4 shows.

Acknowledgements

We thank Manuel Fähndrich for many enlightening discussions about the operation of path simulators and for his insights into context-free reachability. Thanks also to the other members of the Software Productivity Tools Group at Microsoft Research for their input, and to Todd Millstein for his detailed comments on an earlier version of this paper.

References

[AG00] A. Alur and R. Grosu. Modular refinement of hierarchic reactive modules. In *Proceedings of the Twenty Seventh Annual Symposium on Principles of Programming Languages*. ACM Press, 2000.

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[AHM+98] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha : Modularity in model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science, pages 521–525. Springer-Verlag, 1998.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE Computer Society Press, 1988.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97: Concurrency theory*, volume 1243 of *Lecture Notes in Computer Science (LNCS)*, pages 135–150. Springer-Verlag, 1997.

[BR00] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, February 2000.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages*. ACM Press, 1977.

[CC00] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proceedings of the Twenty Seventh Annual Symposium on Principles of Programming Languages*. ACM Press, 2000.

[CR81] L.A. Clarke and D.J. Richardson. Symbolic evaluation methods for program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[DE82] R. B. Dannenberg and G. W. Ernst. Formal program verification using symbolic execution. *IEEE Transactions on Software Engineering*, SE-8(1):43–52, January 1982.

[Dil96] D. L. Dill. The Murϕ Verification System. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 390–393. Springer-Verlag, 1996.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *INFINITY' 97: Verification of Infinite-state Systems*, July 1997.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, 1997.

[HD99] J. Hatcliff and M. Dwyer. Slicing software for model construction. In *Proceedings of the 1999 ACM Workshop of Partial Evaluation and Program Manipulation (BRICS Notes Series NS-99-1)*, January 1999.

[HHK96] R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 423–427. Springer-Verlag, 1996.

[HP96] G.J. Holzmann and D.A. Peled. The State of SPIN. In R. Alur and T.A. Henzinger, editors, *CAV 96: Computer Aided Verification*, Lecture Notes in Computer Science 1102, pages 385–389. Springer-Verlag, 1996.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science, pages 440–451. Springer-Verlag, 1998.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In P. Pfahler U. Kastens, editor, *Proceedings of the 4th International Conference on Compiler Construction (CC'92), Paderborn (Germany)*, volume 641 of *Lecture Notes in Computer Science (LNCS)*, pages 125–140, Heidelberg, Germany, 1992. Springer-Verlag.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers, 1993.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *CAV 97: Computer-Aided Verification*, Lecture Notes in Computer Science 1254, pages 24–35. Springer-Verlag, 1997.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. The British Computer Society, 1971.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[SB92] B. Steffen and O. Burkart. Model checking for context-free processes. In *CONCUR'92, Stony Brook (NY)*, volume 630 of *Lecture Notes in Computer Science (LNCS)*, pages 123–137, Heidelberg, Germany, 1992. Springer-Verlag.

[Sch98] D.A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proceedings of the Twenty Fifth Annual Symposium on Principles of Programming Languages*, pages 38–48. ACM Press, 1998.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[Wei82] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.