On Checking Temporal Properties of Software with Boolean Programs

Jean-Philippe Martin

December 1, 2000

Abstract

This document describes the approach to model checking of software programs suggested by T. Ball and S. Rajamani with their "Bebop" model checker. This approach is novel in that it allows fast checking of terminating programs, even if they include procedure calls or recursion. An extension to this model checker is then described. This extension allows the checking of arbitrary conditions for nonterminating programs, while still supporting procedure calls and recursion. Finally, some open questions are presented.

1 Introduction

Model checking is a very powerful way to prove that a system obeys a given specification. This makes it possible to know with certainty that for example "bad" behavior will never happen, or that some desired property will eventually be reached by the system. Model checking is beginning to be widely used in industry to check hardware designs ([HR00], p. xi).

Currently, software is typically not checked for correctness. Even though software could theoretically be model checked in the same way that hardware is, software systems typically have many more states and this causes what is known as state explosion: the number of states in the model grows very fast and model checking computations become rapidly infeasible.

Several approaches have been proposed to solve this problem with checking software (see "related work" in section 6), but with only limited success so far.

This document describes the approach suggested by T. Ball and S. Rajamani of Microsoft in [BR00], which can handle procedures and recursion – but only for terminating programs with a single entry and exit point.

Their approach is efficient enough to be used in non-trivial programs and it makes it possible to completely automate the checking process, which makes it very attractive and allows even software engineers who are not well versed in logic and math to benefit from the tool.

Their approach works by first abstracting the program in a very radical way, and then refine it progressively in order to be able to verify a certain pre-determined specification Φ using context-free language reachability.

In the first part of this paper I give an overview of this technique and describe it in some detail. In a second part I present an extension of their work that extends the algorithm to

work with (1) graphs with several entry and exit points and (2) to nonterminating programs. Finally, I present a number of open questions for future work.

2 Objectives

[BR00] presents *boolean programs* as a good approach to model checking software because they are abstractions of the original program with less states - therefore avoiding or reducing state explosion. There are a few subtleties with boolean programs but the basic idea is that in these programs, all variables can take only one of two values: true or false. A complete description is given in the next section.

The objective is to use boolean programs to check a C or Java program against a given temporal property Φ. Ultimately, the aim is to be able to check these programs directly and entirely automatically. Theoretically, model checking can be directly applied to the source C program. However, since C has numerical types like "int" that have many possible values, model checking suffers from state explosion and become too computationally expensive. The idea, therefore, is to abstract the original C program into a boolean program, in which the number of states is smaller: the program then becomes easier to model check. Naturally some precautions need to be taken in order for the boolean program to correctly abstract the original program. We discuss these in more details in the next section.

Microsoft's paper presents additional goals, which can be seen as derivations of the original idea. These five goals are as follows:

1. Model checking boolean programs is efficient.
2. If the property is not satisfied in the program, a shortest trace[1] is generated.
3. It is possible to translate other languages such as C++ and Java into boolean programs.
4. Boolean programs can be refined and proven correct, just as FSMs can be refined and proven correct using the semantics of trace inclusion.
5. The model checking algorithms on boolean programs are able to exploit the inherent modularity and abstraction boundaries present in the source programs for more efficiency.

As explained in more detail in the conclusion, most but not all of these objectives have been reached in [BR00]. The ultimate aim is to bring the reliability of hardware to the software world.

The current justification for this research at Microsoft is to make it possible to check device drivers for correctness under Windows 2000. This area is important because device drivers are granted special access privileges into the operating system kernel code and therefore a defective driver could compromise the whole system's integrity (e.g. forcing the user to reboot the machine or compromising the data stored on the disk).

[1] By shortest trace we mean the shortest list of the states (or lines) the program goes through.

3 Boolean Programs

One of the techniques commonly used to combat state explosion when checking a property Φ against a program M is *abstraction*, in which some aspects of M that do not influence the proof are ignored. In [BR00], the authors use boolean programs to iteratively construct abstractions, in the hope that this process helps fight state explosion.

A boolean program is a program written in a restricted form of the C programming language in which all variables are of type boolean. Boolean programs introduce a form of non-determinism, in which one of two branches can be called in a non-deterministic manner (see example below). Recursive procedure calls are possible in this language, as well as the use of pointers. Procedure semantics are always call-by-value. One of the special features of the boolean programs is that they use *"assert" statements* (of the form assert(boolean function)). If the boolean function evaluates to false, then this path is pruned. These assertions are not inserted by the programmer directly but rather by the an earlier phase of the checking process itself. We discuss these statements in more detail in section 4.3. The complete grammar of boolean programs is presented in annex A. Here is an example of a program with the corresponding boolean program:

Original Program	*Boolean Program*
procedure main()	main()
begin	begin
b := 3	skip;
if (a==5) begin	if (*) then
printf("five")	skip;
end	fi
A()	A();
	skips
end	end

As you can see, most instructions are replaced with "skip" statements (that do nothing). Also, the condition of the if branch has been replaced with a nondeterministic choice. The procedure call stays, however. For technical reasons, a skip statement is inserted after all such calls. It is clear in this example that all lines that can be reached in the original program are also reachable in the boolean one.

These boolean programs can also be described with a grammar and represented as graphs. Because of the procedure calls and recursion, however, these graphs are not as simple as finite state machines and the conventional model-checking algorithms cannot be directly applied to them.

Boolean programs define context-free languages which can be checked by a push-down machine. Boolean programs build upon work by [RHS94], which defines the interprocedural, finite, distributive, subset problem (IFDS) framework, a variant of Sharir and Pnueli's "functional approach" to interprocedural dataflow analysis. The extension allows them to handle local variables and parameters.

Boolean programs are represented using a variant of RHS' IFDS framework. In this framework a program is represented as a directed graph $G^* = (N^*, E^*)$ called a *supergraph*. G^*

consists of a collection of graphs (one for each procedure), one of which, G_{main}, represents the program's main procedure (i.e. contains the program's main entry and exit points). Each graph G_i has a unique start node s_i and a unique exit node e_i. The other nodes in the graph represent the statements and predicates of the procedure in the usual way, except that a procedure call is represented by two nodes, a *call* node and a *return-site* node.

In addition to the ordinary intraprocedural edges that connect the nodes of the individual graphs, for each procedure call, represented by call-node c and return-site r, G^* has three edges:

- An intraprocedural *call-to-return-site* edge from c to r;
- An interprocedural *call-to-start* edge from c to the start node of the called procedure
- An interprocedural *exit-to-return-site* edge from the exit node of the called procedure to r.

The call-to-return-site edges are included so that the IFDS framework can handle programs with local variables and parameters. The dataflow functions on call-to-return-site and exit-to-return-site edges permit the information about local variables that holds at the call site to be combined with information about global variables that holds at the end of the called procedure.

These supergraphs represent context-free languages because procedure return semantics are enforced. These language semantics require procedure calls to eventually return to the calling site: each procedure call (call-to-start edge) must be closed by a corresponding procedure return (exit-to-return-site edge), and these pairs must be properly nested. The paths that satisfy these requirements are called *(interprocedurally) valid paths.*

The paper makes the unformulated assumption that these programs always terminate but in a later section I show an extension of bebop that handles nonterminating programs.

4 Algorithm

4.1 Motivation

[BR00] presents an algorithm to determine whether a certain set of error states can be reached in a boolean program. It is argued that this is sufficient to prove any temporal property Φ of boolean programs because the original boolean program B can be extended with a finite state machine that checks for the complement of the temporal property Φ. We obtain a new boolean program B' in which some states S' correspond to a violation of Φ. Proving that Φ is an invariant of C is equivalent to proving that S' is not reachable in B'. This reachability analysis is then performed by exhaustively exploring all paths. In other words, the property is checked by exhaustively checking all possible paths for a counter-example.

The main contribution of the paper is the idea of automatically generating abstractions of the original problem. Conceptually, the problem is first abstracted as much as possible and then progressively refined until being "close enough" to the original program - but still, hopefully, much simpler.

Another important point is that the algorithm is efficient because it creates "summaries" of procedures and then reuses them when the procedure is called again.

4.2 Iterative Refinement

Since the original program C is too big to be model checked directly, it is *abstracted* by a boolean program B_0 in the following manner:

1. All conditions are replaced by a non-deterministic choice.
2. All instructions are replaced by a skip operation (which does nothing).

This first abstraction is called the *boolean-normal form* of C. There is a one-to-one mapping from the statements in C to the statements of B_0. B_0 is an abstraction of C in the sense that each valid path in C corresponds to a valid path in B_0. However we say that B_0 is *weaker* than C because valid paths in B_0 do not necessarily correspond to valid paths in C (maybe in C it appears that the condition in the "if" statement cannot be satisfied). It follows from this that if a statement is reachable in C then it is also reachable in B_0. Reciprocally, if a statement is not reachable in B_0 then it is not reachable in C either. We use the notation $C \preceq B_0$ to denote this fact (Intuitively, this notation means that the set of valid traces in B_0 is a superset of the set of valid traces in C).

To determine whether a given statement s (witness of a violation of some temporal property Φ) is reachable in C, the algorithm first builds B_0 in the manner described above. If s is not reachable in B_0 then we know it is not reachable in C either and we are done. If, on the other hand, s turns out to be reachable in B_0 then a path is built from the witness trace and used in a path simulation on C. If the simulation succeeds then we know that s is reachable in C, and we have a sample trace that violates Φ. If not, then we use the information of the trace to come up with a new abstraction B_1 that is more accurate: $C \preceq B_1 \preceq B_0$. All realizable paths in C are still realizable in B_1, but the offending trace (realizable in B_0) is no longer realizable in B_1. We say that B_1 is a *refinement* of B_0.

The algorithm then tries once again to reach s, in B_1 this time. There are three different ways the algorithm can terminate:

1. The statement s is determined to be reachable (if necessary by refining until B_n is nearly indistinguishable from C), and a witness trace is produced.
2. The statement s is determined to not be reachable in C.
3. s is reachable in one of the abstractions B_i but the algorithm is not able to create an appropriate refinement of B_i. In this case, the algorithm terminates with an answer of "don't know".

4.3 Refinement

In this section, I describe in more detail how a boolean program B_i is refined into B_{i+1}. This refinement consists of two steps. First, an offending trace in B_i is used in a path simulator to come up with the variables that are be added to B_i. Second, the variables are added and other modifications are made to generate a new boolean program B_{i+1} which is both an abstraction of C and a refinement of B_i.

The first step is described in section 4.6, "Path Simulation". In short, it consists in running through the program step by step, taking note of the various assertions in order to detect an inconsistency.

Suppose that B_i's trace corresponds to an (invalid) path in C in which some numerical variable v is first set to 5 and then compared with 2. The path is feasible in B_i because the test is replaced by a nondeterministic choice, but it is not feasible in C because it is not possible that $v = 5 \wedge v < 2$. In this case the path simulator would determine that a new boolean variable $\{v < 2\}$ is necessary ("$\{v < 2\}$" is not an expression but the variable name itself). That variable is introduced in B_{i+1} and set to 0 (false) at the location corresponding to the assignment in C. The nondeterministic branching that corresponds to "$if\ v < 2$" in C is *not* modified, but instead the "yes" branch is be annotated with $assert(\{v < 2\})$ and the "no" branch is be annotated with $assert(not\{v < 2\})$. All other locations in B_i where the value of $\{v < 2\}$ is modified or tested are updated as well.

One special case occurs when the value of the variable modeled by $\{v < 2\}$ is modified, but the new value cannot be determined (for example, it depends on another variable of C that has not yet been added to B_{i+1}). In this case, $\{v < 2\}$ is set to a special value "*", which in effect means "don't know". The rules of boolean logic are expanded so that any operation on "*" returns "*", and $assert(*)$ does not result in a branch being discarded. In the case where $\{v < 2\}$ is not set to "*", the assertion statements that have been inserted after the nondeterministic branch will cause the "wrong" path, when taken, to be immediately discarded. The end result is identical to the replacement of the nondeterministic branch by a deterministic one [2]. I can only guess that the reason for this design decision is that the authors expect that "assert" statements may be inserted for some other reason as well.

4.4 Reachability

It is clear that state exploration on a finite state machine can discover counterexamples to any temporal property Φ. The paper claims that their algorithm ("Bebop") works not only for finite state machines, but also for context-free languages. Furthermore, the algorithm not only finds counterexamples but can also prove that none exist.

Let us deal with the second claim first. The reason why Bebop can prove that no counterexample exists is because it exhaustively explores all the states. Clearly, if no reachable states violates Φ then $M \models \Phi$.

But how can there be only finitely many states if the program can be recursive – and, hence, its runs can last infinitely long? Well, the program may run infinitely long, but since all variables are defined over finite domains, it only explores a finite number of states. These states are defined as a tuple containing the value of all the variables in the system and the number of the line that is about to be executed[3]. For example if Σ is the set of lines in the program, and Δ is the set of boolean variables then the state space is $\Sigma \times 2^{\Delta}$. There is a finite number of such states and, since they only need to be explored finitely often, model checking will terminate.

Therefore, it is possible to detect unreachable states and prove temporal properties Φ. By

[2] Branch that would revert to being nondeterministic if the value of the variable turned out to be "*".

[3] Note that this is different from the traditional notion of state, which also includes the call stack.

the same argument, it is possible to check whether the program eventually terminates or not (despite the fact that in all generality this problem is undecidable).

4.5 Summary Edges

The algorithm attempts to take advantage of the abstraction boundaries present in C. To this effect, while model checking one of the boolean programs B that abstracts C, it computes *execution summaries* for each procedure call. These execution summaries can be used directly when the same procedure is called again, instead of having to go through the procedure subgraph. This technique is described in [BR00c].

The execution summary, for each procedure p, contains a set of pairs of the form (I, O). I is a valuation for global variables when entering p and O is the valuation for these same variables just after the procedure call. For example, if in a particular execution the variable $\{a < 2\}$ is true, p is called and this variable is false after the call, then a pair (I, O) is be added to the set of summaries of p and $\{a < 2\} \in I$ and $(not\ \{a < 2\}) \in O$. An entry for each of the other global variables is added to I and O, each valued to either 'true', 'false' or '*'.

The goal of execution summaries is to speed up checking computation for realistic size programs, which is certainly reasonable. The algorithm as described exploits the modularity from procedural abstraction. Other abstractions like the ones introduced by object orientation or package boundaries, however, are not utilized.

4.6 Path Simulation

Path simulation consists in stepping through the program, following the trace generated on the corresponding graph. This is done after a trace has been found that reaches a state of Φ, and has two goals: (1) Determining whether that run is valid in C and (2), if it is, identifying the elements which will then be incorporated into the boolean program B_i to refine it and continue the process of model checking.

The path simulator goes through C, following the trace τ that was originally found on B_i. This is possible because, again, there is a one-to-one correspondence between the states in C and the states in B_i. Path simulation is based on symbolic evaluation, described in [CR81, DE82].

A path simulator contains an environment (Env) mapping variables to symbolic values and a set of conditions ($Cond$) over these symbolic values. Both Env and $Cond$ are updated as the simulator steps through the program. If $Cond$ never becomes self-contradicting then τ was effectively a trace of C.

Otherwise, the infeasibility of the path can be "explained" as a conjunction of conditions E derived from the state of the path simulator where the contradiction arises. The details of how this procedure are given in [BR00b].

The technique is inspired from the concept of *strongest precondition* and a simplified explanation follows: Let α be the statement in which $Cond$ becomes self-contradictory. Since only these two kind of statements that modify $Cond$, α is either an assignment of the form $a = b$ or an assertion of the form $assert(c)$. In the first case, the contradiction is created through some assertion $assert(c)$ in $Cond$ that is not compatible with the new value of a.

In both cases, the expression represented by c is violated in C but not in B_i. Let $c = \bigwedge_i v_i$. Adding the variables $\{v_i\}$ to B_i yields a new program B_{i+1} in which α is no longer reachable via this path. Furthermore, $P \preceq B_{i+1} \preceq B_i$.

Let us go through an example, with the following program C (pseudocode, numbered lines). The goal is to determine whether the ERROR statement is reachable in C.

```
1: a := 1
2: if (a>5) then
3:  ERROR
4: end
```

First boolean program B_0:

```
1: skip;
2: if (*) then
3:   skip;
4:   ERROR
5: fi
```

The ERROR state is reachable in B_0. The path simulation, however, will determine that this path is not valid in C because a:=1 is not compatible with the a>5 mandated by this *if* branch. The failing condition c is: "a>5". Therefore a new variable $\{a > 5\}$ is introduced, resulting in the following program for B_1:

```
0: decl {a>5};
1: {a>5} := F;
2: if (*) then
3:   assert({a>5});
4:   ERROR
5: fi
```

Exhaustive search now reveals that state 4 is not reachable in B_1, which concludes the proof that ERROR (vertex 4) is not reachable in C. In fact, the complete algorithm has the further optimization that it does not use all the expressions in *Cond* but possibly only a subset of them.

5 Open Questions

In the course of this project, a number of candidate "open questions" have been asked. I will present them here, even though some of them have been answered in the meantime.

5.1 Liveness

One of the open questions I presented in the status reports was, "Can this algorithm check liveness properties?". It turns out that it can.

Properties like AGf ("it is always the case that f") can be checked by trying to reach statements that would cause f to be violated. If after exploring all paths no such statement has been reached then the property holds.

Liveness properties are of the form AFf ("inevitably f") or AG(f $\Longrightarrow$ AFg) ("every time that f, g eventually happens"). These properties can be checked just like safety, by adding code that checks whether the complement of these properties happens and by checking to see whether one of the states witnessing this violation is reachable. For example in the case of AFf, a boolean variable $\{f\ happened\}$ would be created and initially set to false. Every statement in which f indeed happens be modified to add $\{f\ happened\} = true$. Finally, just before main's exit node we add the command: "$if\ (not\ \{f\ happened\})\ then\ \Psi$" and check for reachability of Ψ. If Ψ is reachable, then the liveness property has been violated and the algorithm produces a trace that witnesses this violation.

Φ=AG(f $\Longrightarrow$ AFg) can be checked in a similar way, by keeping track of f and g and then checking the reachability of the exit node with f true and g false. If it is not reachable, then Φ holds.

So we can indeed check liveness, the only constraint being that we identify where the "end" of the program is.

5.2 Multiple entry or exit points

Allowing multiple entry points can be handled almost trivially by creating a new entry point with an edge to each of the previous entry points. This new vertex is equivalent to a nondeterministic choice of any of the entry points. The exit points can then be similarly linked to a single new exit point.

5.3 Non-Terminating Programs

It turns out that, even though the paper does not mention this fact explicitly, the algorithm it describes can be extended to non-terminating programs (i.e. ω-Push Down Automata). Additionally, this extension makes it possible to check liveness for non-terminating programs as well.

Until this point we assumed that the program contained an entry point and an exit point. We shall first relax the second assumption and do without an exit point. Rather, the exit point is still there but it needs not be reachable.

Safety can still be checked in the usual way, even in this framework: the model checker will visit all states in a search for a violation of the liveness property. If no such violation happens, then the program has been proven safe.

As discussed in the previous paragraph, the same reasoning applies to liveness, in the sense of "eventually F" (AFF). However, in non-terminating programs, a much more interesting liveness property arises: $A\overset{\infty}{F}F$ (infinitely often F, or AGAFF). Can we extend the boolean program algorithm to allow checking for this kind of liveness properties?

Remember that the exploded supergraph described in [RHS94] contains a node per program state and edges represent valid transitions. Is it enough to directly model check this graph?

Not quite, because interprocedural paths are a concern.

It turns out that this checking is nevertheless possible after some modifications to both the Bebop and the dataflow analysis algorithm. We use EG, EU and EX as our base for model checking. It is well known that all CTL formulas can be expressed using only these three temporal operators.

Before I can explain the algorithm, a few definitions need to be introduced.

Definition 1 *A* statement *is an instruction in a boolean program B. Each such statements corresponds to a* vertex *in the dataflow supergraph V_B that corresponds to B. Intuitively, edges will be introduced between two vertices if the corresponding instructions are in succession.*

Definition 2 *For a set V of variables in the boolean program B, a* valuation Ω *to V is a function that associates every boolean variable in V with a boolean value.*

Definition 3 *A* state *of B is a pair $\langle i, \Omega \rangle$, where $i \in V_B$ and Ω is a valuation to the variables in $InScope_B(i)$. Intuitively, a state contains the program counter(i) and the value of all the variables visible at that point (Ω).* States(B) *is the set of all states of B.*

Definition 4 *A* path edge *incident into a vertex v is a pair $\langle \Omega_1, \Omega_2 \rangle$ where Ω_1 is a valuation of the initial vertex that allows the procedure which contains v to be reachable (the weakest precondition of $ProcOf_B(v)$) and Ω_2 is a valuation that can occur at v.* PathEdges(v) *is the set of all path edges incident into v.*

The first step of the new algorithm is to create the exploded supergraph. This is done as follows.

1. Run the original Bebop algorithm once, creating all the summary edges and path edges.
2. Transform the dataflow supergraph V_B into the exploded supergraph E. Formally, for each vertex $v \in V_B$, create x nodes v'_i, $x = |\text{InScope}_B(v)|$. Each of these nodes correspond to a state.
   For each edge e from v to $w \in V_B$, create an edge e'_{ij} between each v'_i and w'_j for which $\exists(\langle \Omega_1, \Omega_2 \rangle \in Join(PathEdge(v), transfer_v), \langle \Omega_3, \Omega_4 \rangle \in PathEdge(w)) : \Omega_2 = \Omega_4$ (i.e. state w'_j is a successor of state v'_i — being at vertex v with state i it is a valid transition to move to vertex w with state j).
   Note that this will create summary edges in E, too. We call summary edges in E all the edges that link two nodes that were linked with a summary edge in V_B.

Note that this step has to be performed only once. The different temporal properties are then checked on this graph.

To check $E[\psi_1 \mathrm{U} \psi_2]$:

1. Mark all states of ψ_2 with "$E[\psi_1 \mathrm{U} \psi_2]$".
2. Repeat until no change:

3. label any state with $E[\psi_1 \mathrm{U} \psi_2]$ if it is labeled with ψ_1 and at least one of its successors (excluding summary and exit-to-return-site edges) is labeled with $E[\psi_1 \mathrm{U} \psi_2]$.

4. If a state n_i is labeled with ψ_1 and all its successors via call-to-start edges are labeled with $E[\psi_1 \mathrm{U} \psi_2]$, and if its successors via all call-to-return-site edges are also labeled with $E[\psi_1 \mathrm{U} \psi_2]$ then label n_i with $E[\psi_1 \mathrm{U} \psi_2]$.

5. end repeat

After the algorithm completes (which will eventually happen since the number of states in finite), we can say that $E[\psi_1 \mathrm{U} \psi_2]$ holds for B iff at least one of the states in E that corresponds to $\mathrm{First}_B(\mathbf{main})$ is labeled $E[\psi_1 \mathrm{U} \psi_2]$. If $E[\psi_1 \mathrm{U} \psi_2]$ does not hold for B, it does not hold for C either.

To check $EG\psi_1$:

1. Mark all states of ψ_1 with "$EG\psi_1$".

2. Repeat until no change:

3. label any state with $EG\psi_1$ if it is labeled with ψ_1 and all of its successors (excluding exit-to-return-site edges) are labeled with $EG\psi_1$.

4. If a state n_i is labeled with ψ_1 and all its successors via call-to-start edges are labeled with $EG\psi_1$, and if its successors via all call-to-return-site edges are also labeled with $EG\psi_1$, then label n_i with $EG\psi_1$.

5. end repeat

After the algorithm completes we can say that $EG\psi_1$ holds for B iff at least one of the states in E that corresponds to $\mathrm{First}_B(\mathbf{main})$ is labeled $EG\psi_1$. If $EG\psi_1$ does not hold for B, it does not hold for C either.

To check $EX\psi_1$:

1. label any state with $EX\psi_1$ if it has at least one successor (excluding summary edges) labeled with ψ_1.

Because we run Bebop beforehand, exit nodes of functions are be marked with $EX\psi_1$ if ψ_1 holds in the return node of at least one of the places which called that function. This is the expected behavior. Again, if $EX\psi_1$ does not hold for B it does not hold for C either.

Checking for ψ_1, $\psi_1 \vee \psi_2$ and $\psi_1 \wedge \psi_2$ is done in the traditional model checking manner, since these properties are not concerned with transitions.

Note that this algorithm exploits the abstraction boundaries created by the interprocedural calls. If it determines that B does not satisfy the property Φ (expressed in terms of EG, EX and EU), then we know that C does not satisfy it either: We proved the program wrong and can produce a counterexample trace.

If, however, the property Φ holds then it may or may not hold in C. To determine this, Bebop uses a witness trace that satisfies Φ and try to apply it to C. If the trace works for

C then the program was proved correct. Otherwise, Bebop uses the trace to come up with a refined version of B, B_{i+1}, which will be model-checked similarly.

Just like the original algorithm, this modified version can terminate in one of three ways:

1. The statement s is determined to be reachable (if necessary by refining until B_n is nearly indistinguishable from C), and a witness trace is produced.
2. The statement s is determined to not be reachable in C.
3. s is reachable in one of the abstractions B_i but the algorithm is not able to create an appropriate refinement of B_i. In this case, the algorithm terminates with an answer of "don't know".

Unlike the original Bebop algorithm however, this modified version is able to check programs that do not terminate. This can prove directly useful when checking certain kind of device drivers, which was part of the original goal of the Bebop effort. We shown that checking of nonterminating programs was possible, but work remains to try to find faster algorithms.

5.4 Other open questions

Luckily, I have not been able to answer all the open questions I came up with and therefore the following questions remain open:

1. Can these techniques be expanded to use other programming languages abstractions such as classes and modules?
2. Can these techniques be expanded to take concurrent programs into account?
3. Is it possible for some past refinement to become redundant relative to a newer refinement, and if so can the algorithm be enhanced by removing such redundant state information?

The answer to this first open question is almost certainly yes; it will be interesting to see how this is done. The reason why these abstractions are not leveraged is probably that the primary target language is C (which is not object-oriented). I have no doubt, however, that Microsoft will eventually extend their work to include other languages like C++ or Java and then find a way to leverage further abstractions since (1) this will enable model checking of larger programs and (2) the extension will be natural and should be fairly easy to implement.

One way to leverage object orientation is to look at variables: Objects contain variables that are not global to the whole program nor local to a procedure call, but local within the group of procedures that comprise the object. Therefore, if no procedure of an object is ever called then its variables can safely be eliminated from the model. Another idea would be to utilize object invariants: objects are typically designed to have invariants, and sometimes code is inserted in order to detect invariant violations. Since the information is available, a model checking tool could be developed which model checks these invariants (and hence formally establishes their invariance) and uses them to speed up the checking of the complete program.

A similar approach can be taken with packages, or modules. Additionally, modules are (or should be) usually largely independant of each other, which suggest that they could be checked separately, using some new mechanism similar to procedure summaries to combine the results of the individual checks. Also, the results of these checks could be stored on persistent medium so that if the engineer modifies one module and then verifies the software, the untouched modules do not have to be explored at all.

The second question is an interesting one, because it turns out that after I included it in this paper Microsoft has come up with a solution to it! This solution is described in a paper by Thomas Ball, Sagar Chaki and Sriram K. Rajamani ([BCR00]). The abstract describes their work as: "We present Local/Global Finite State Machines (LGFSMs) as a model for a certain class of multithreaded libraries. We have developed a tool called Beacon that does parameterized model checking of LGFSMs". Furthermore, they have used this tool to check a critical safety property of Rockall.

The third open question hints at one of the possible limitations of the bebop algorithm: are all of the refinements necessary, of will the refinement process introduce unnecessary complexity of its own?

A solution to this question would involve an in-depth exmination of the algorithm, or perhaps simply luck at finding "worst-case" examples. These examples may, in turn, provide insight into ways to enhance the boolean program algorithm.

6 Related Work

Applying model checking to software has been explored before, and several notions of refinement, including trace containment, have been studied in the past [Mil71, AL88, Kur94]. Boolean programs can be viewed as abstract interpretations of the underlying program [CC77]. The connection between model checking, dataflow analysis and abstract interpretation have been studied before [Sch98, CC00]. The model checking problem for pushdown automata has been studied before as well [SB92, BEM97, FWW97].

The CFL reachability framework from [RHS94] builds on earlier work in interprocedural dataflow analysis from [KS92] and [SP81]. Exploiting design modularity in model checking has been recognized as a key to scalability of model checking [AH96, AG00].

Program slicing is a projection operation on programs that preserves the execution behavior of the projection [Wei82]. However, as explained in [BR00], it is different from the iterative refinement of boolean programs.

7 Conclusion

Looking back at the goals of the [BR00] paper, we can see that three of them have already been reached (even though the last one is open to debate):

1. If the property is not satisfied in the program, a shortest trace is generated.

2. Boolean programs can be refined and proven correct, just as FSMs can be refined and proven correct using the semantics of trace inclusion.

3. Model checking boolean programs is efficient.

And two have not:

1. It is possible to translate other languages such as C++ and Java into boolean programs.

2. The model checking algorithms on boolean programs are able to exploit the inherent modularity and abstraction boundaries present in the source programs for more efficiency.

Even though it is certainly possible, the translation of other languages into boolean programs has not been implemented yet. Also, even though the bebop model checker takes advantage of the abstraction of procedures, it does not (yet) take advantage of object orientation or the other existing abstractions.

We have been able to show extensions of the algorithm that widen its applicability to nonterminating programs. These extensions also take advantage of the interprocedural abstractions and are also able to automatically come up with an efficient abstraction of the initial program. A number of open questions have then been discussed, and suggestions or ideas for future work have been formulated.

This paper and many similar ones before it tries to make it possible to check software against a given model, I would say despite the language used (C in this case). I believe that in the long run the only way to come to a really useful checking tool is to modify the language itself to leverage the power of model checking or other approaches. To quote Dijkstra: "A programmer has to be able to demonstrate that his program has the required properties. If this comes as an afterthought, it is all but certain that he won't be able to meet this obligation" (EDW1305). This can be read as saying that programmers should be mathematicians and it certainly would help if they were. However a much more practical approach would be to give programmers powerful tools with which they can check their programs as they write them, not as an afterthought. This should certainly make checking both more efficient and more productive, and maybe would incitate some software engineers to take some classes in math or formal verification!

References

[AG00] A. Alur and R. Grosu. Modular refintments 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.

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

[BCR00] Thomas Ball, Sagar Chaki and Sriram K. Rajamani. *Parameterized Verification of Multithreaded Software Libraries*. Software Productivity Tools, Microsoft Research, 2000.

[BR00] Thomas Ball, Sriram K. Rajamani. *Checking Temporal Properties of Software with Boolean Programs*. Software Productivity Tools, Microsoft Research, 2000.

[BR00b] Thomas Ball, Sriram K. Rajamani. *Boolean Programs: A Model and Process For Software Analysis*. Technical Report MSR-TR-2000-14, Microsoft Research, 2000.

[BR00c] Thomas Ball, Sriram K. Rajamani. *Bebop: A Symbolic Model Checker for Boolean Programs*. Software Productivity Tools, Microsoft Research, 2000.

[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.

[CR81] L.A. Clarke and D.J. Richardson. Symbolic evaluation methods for program analysis. In S.S. Muchnick an dN.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 Transaction on Software Engineering*, SE-8(1):43-52, January 1982.

[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.

[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.

[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.

[HR00] M. Huth and M. Ryan, *Logic in Computer Science: Modeling ans reasoning about system*, Cambridge University Press.

[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.

[RHS94] Thomas Reps, Susan Horwitz and Mooly Sagiv. *Precise Interprocedural Dataflow Analysis via Graph Reachability*. University of Wisconsin, 1994.

[SB92] B. Ste en 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 ow 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 dataflow 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.

A Grammar of Boolean Programs

Syntax	Description
prog ::= *decl** *proc**	A program is a list of global variable declarations followed by a list of procedure definitions
decl ::= **decl** *id*$^+$ **;**	Declaration of variables
id ::= [**a-zA-Z_**] [**a-zA-Z0-9_**]* | **{** *string* **}**	An identifier can be a regular C-style identifier or a string of characters between '{' and '}'
proc ::= *id* **(** *id** **) begin** *decl** *sseq* **end** *sseq* ::= *stmt*$^+$	Procedure definition Sequence of statements
lstmt ::= *stmt* | *id* **:** *stmt*	 Labeled statement
stmt ::= **skip ;** | **print (** *expr*$^+$ **);** | **goto** *id* | **return;** | *id*$^+$ **:=** *expr*$^+$ *id* | **if (** *decider* **) then** *sseq* **else** *sseq* **fi** | **while (** *decider* **) do** *sseq* **od** | **assert (** *decider* **);** | *id* **(** *expr** **);**	 Parallel assignment Conditional statement Iteration statement Assert statement Procedure call
decider ::= ***** | *expr*	Non-deterministic choice
expr ::= *expr binop expr* | **!** *expr* | **(** *expr* **)** | *id* | *const*	
binop ::= '|' | '**&**' | '$\wedge$' | '**=**' | '**!=**' | '$\Longrightarrow$'	Logical connectives
const ::= **F** | **T**	False/True