A Verifier for Temporal Properties

David Mandelin

May 3, 2002

1 Background

Before distributing a program, programmers would like to know for certain that it does what it is intended to do. The only way to do this is to write a proof, which is infeasible for real programs.

Given this, programmers usually perform validation testing, in which they run the program with certain inputs and verify that the program yields the correct output for those inputs. Testers attempt to find inputs for which the program runs incorrectly. Failure to do so is taken as evidence that the program is correct. However, validation testing cannot prove anything.

In addition to validation testing, programmers also use compilers that check for certain kinds of errors. For example, many compilers report a warning if a variable is used before it is initialized. The behavior of the use is undefined, so it is very likely that the program is incorrect. The compiler can be understood as verifying that the program has a certain property, the property that all variables are used only after initialization.

Note that the program may be written with an uninitialized variable that does not affect the output. This is called a benign property violation.

The property under discussion is called a temporal property because it refers to the order in which certain events take place. Automatic verification of temporal properties is an active field of programming language research. The goal is to specify temporal properties that programs must have to be correct, and then automatically check them.

Libraries and APIs are a major focus of temporal property verification. Libraries and APIs often have many rules that users must obey, and many of these rules are temporal. For example, in the C stdio library,

A file must be opened before it can be read.

This rule is easy to understand, and has a compact formal representation, as shown in Figure 1. Also, verifiers have been implemented that can check such rules. Therefore, checking APIs is a promising area of research.

In formal terms, a verifier takes a temporal property T and a program P as input, and attempts to prove that P has property T. If successful, it reports "yes". If unsuccessful, it reports a counterexample. Typically, the verifier is sound, so that it reports that P has property T only if P really does have property T. Since these properties are undecidable, the verifier is necessarily incomplete. Any such verifier will sometimes report that P does not have property T when it really does, or that it does not know whether P has property T, or it may fail to terminate.

Note that automatic verification has an important advantage over testing: it can prove that a program is free of certain kinds of errors, which validation testing can never do.

2 Introduction

Temporal property verification is an active field of research, and several verifiers have been implemented. This verifier, YASC, does not promise anything particularly new, but it will be valuable for several reasons.

- Strauss [ABL02] is a specification miner, a tool which partially automates writing temporal property specifications. YASC will be used in experiments to demonstrate how Strauss can be used to mine specifications and then find bugs.
- Each property verifier uses its own specification language. YASC will have its own language and will provide experience with the virtues and pitfalls different specification languages.
- Property verification attempts to verify every path in the program, which is in general NP-hard. There are verifiers that use polynomial-time approximations, but there is still work to be done in this area.
- The design of YASC is based on that of ESP [DLS02], a scalable temporal property verifier. ESP works only on specifications with one variable. Future work may be able to extend YASC to work with multiple values.

The initial target is a verifier that can check properties that involve only one variable intraprocedurally. Future work may include interprocedural analysis and verification of properties with more than one variable.

3 The Specification Language

3.1 Introduction

A property verifier must have a language for expressing the temporal properties, or *specification language*. The specification language determines what properties can be expressed, how easy it is to express them, and how easy it is to understand them. It also influences the design of the verifier.

Often, specification languages are based on NFAs that accept sequences of events that violate a property. Consider the following informal rule for the C stdio library:

> A file must be opened before it can be read or closed, and it cannot be used in any way after it has been closed.

The NFA in Figure 2 encodes this rule. A sequence of events is checked by feeding it into the NFA. If the NFA reaches the FAIL state, the sequence violates the rule. Otherwise, it is valid. Note that any event that does not appear on an edge is ignored. For example, calls to fwrite are ignored.

This NFA, however, expresses the rule only for programs that use only one file. Consider this sequence of calls:

```
FILE * f1 = fopen(``foo.txt'', ``r'');
FILE * f2 = fopen(``bar.txt'', ``r'');
fclose(f1);
fclose(f2);
```

This is a correct use of the stdio library, but the NFA in Figure 2 reaches FAIL. The NFA needs to be extended to work with any number of open files. Since the number of open files is (conceptually) unbounded, the resulting specification is not truly a finite automaton, but most authors call it an NFA, and that practice will be followed here.

Figure 3 shows an extended NFA for the informal rule. This NFA has events that are parametrized by a value, f. Thus, sequences of events are defined by values that flow among them.

One way to view this NFA is as a universally quantified version of the NFA in Figure 2. For each value X of type FILE * that is created during execution, the set of calls to fopen, fread, and fclose using X must occur in the sequence given by the NFA.

This view is conceptually very clear, but it is not obvious how a verifier would determine the set of all such values X. Another way of viewing the NFA is to view the edges as patterns. Every time an event occurs during execution that matches a pattern on an edge leading out of START, a new instance of the NFA is created with the variable in the NFA is bound to that value. For example, if a program calls fopen, which returns the value 0xffe81010, then a new instance of the NFA is created with the binding y = 0xffe81010. This binding is used to substitute for y everywhere in the new instance of the NFA, so the new instance changes state only when the value 0xffe81010 appears in a call. If any instance of the NFA reaches FAIL, then the program violates the specification. Using this view, it is clear that the verifier can search for events that cause a binding, and thus determine all of the values created by the program.

The extended NFA may contain multiple variables. In that case, any event that matches an edge from the current state that has an unbound variable creates a new binding. This is a direct generalization of the single-variable case.

3.2 YASC Specification Syntax

Here is how the NFA in figure 3 is encoded as a YASC specification:

```
y = fopen()
        START open
fread(_, _, _, y)
        open open
        closed FAIL
fclose(y)
        open closed
        closed FAIL
```

```
\$exit(y)
        open FAIL
```

The unindented lines are patterns that represent edge labels. The pattern is similar to the C function call syntax. The variables are specification variables, which are bound to values. The underscore is a placeholder indicating arguments to be ignored. Not all of the arguments need be matched. If the pattern contains fewer variables than the C function, the other variables are ignored.

The special pattern \$exit(y) matches the special *exit* event. Conceptually, the exit event occurs at the end of the program. This allows the verifier to check for resource leaks. The pattern refers to the variable y because each value is tracked independently.

Following each pattern is an indented list of edges to be labeled with that pattern. An edge is specified as a pair of states. A state may be named as any legal C identifier. The set of states is inferred from the edge names.

There are two special state names. START is the start state. FAIL is the failure state. If the verifier finds a program execution path that reaches FAIL, it will report an error.

3.3 YASC Specification Semantics

The verifier must operate on an input program and specification (P, S) as if doing the following:

First, it forms P', the product of P and S. Informally, P' works like P, but also maintains the state of S, the NFA, as it undergoes transitions caused by events in P. Then, it tries to prove that P' never reaches the FAIL state. If it can prove that P' never reaches FAIL, the verifier prints out "Correct". Otherwise, it prints a path through P, which should reach FAIL. However, since the verifier is not complete, the path may not actually reach FAIL.

Formally, P' runs as follows: It starts the NFA of S in the START state. Then it runs P normally. On each function call, it tries to match the call against all patterns on edges leading out of the START state. If there are no matches, it continues running P. If there are matches, for each match in binds the variable in the specification to the corresponding value in the function call in the program. Then it continues executing P nondeterministically along each branch. It also updates the state along each branch to the state following the edge.

On each function call, it tries to match the call against all patterns on edges leading out of the current state. If there are matches, it nondeterministically follows all branches, updating the state accordingly.

4 Verification Algorithms

The algorithm used by YASC is based on that used by ESP, another verifier. This algorithm is designed for specifications with one variable. The algorithm is developed below in a sequence of increasingly powerful versions.

4.1 One Value, One Procedure, All Paths

The simplest problem is to verify a program with only one value for the specification variable and only one procedure. The problem can be solved as a data-flow analysis problem. The problem is to find the set of states that the NFA can possibly be in at each program point. It can be solved using the following data-flow equations:

$$in(entry) = \{START\}$$

For a function call node in the CFG,

$$out(n) = \{s' | s' = nextstate(s, n) \wedge s \in in(n)\}$$

The exit node is handled the same way if there are $exit events in the specification.

For a merge from nodes m_1 and m_2 to n,

$$in(n) = out(m_1) \cup out(m_2)$$

This problem is easily solved using standard data-flow analysis. Note that since the merge function is union, this analysis assumes that either direction may be taken for any branch. Thus, this analysis follows all paths, whether feasible or not, which may cause it to report many spurious errors.

4.2 Many Values

This version extends the previous problem by allowing many values for the specification variable. It can be solved by first finding all calls in the program that match a binding edge in the specification.

Next, the verifier performs a value-flow analysis to find all the possible values for each variable in the program. Using this, it can take a slice of the program for each variable, and then perform the data-flow analysis for each slice.

4.3 Many Procedures

The verifier can be extended to work for many procedures using standard interprocedural analysis techniques. Both the value-flow and data-flow analyses are performed interprocedurally.

4.4 Feasible Paths

The verifier can be made more accurate, meaning that it reports fewer spurious errors, if it considers only feasible paths. However, doing so requires a full simulation of the program, which is too expensive. ESP presents a method of efficiently approximating feasible paths.

Briefly, ESP tracks both the NFA state set and part of the program state. When a branch is taken, it captures the differences in program state along the two branches implied by the branch predicate. For example, if the NFA state set is $\{START\}$, and it reaches a branch "if (flag) ...", the state is $\{(START, flag)\}$ along the true branch and $\{(START, \neg flag)\}$ along the false branch.

At a merge, instead of performing a union, ESP performs a union and then partitions the states by their NFA state. Within each state, it merges the program state. If, in the previous example, the two branches merged without any other state changes, they would be merged into

$$\{(START, flag \vee \neg flag)\}$$

which reduces to $\{START\}$. However, if the true branch opened a file while the false branch did not, the state would be

$$\{(open, flag), (START, \neg flag)\}$$

In this way, ESP keeps track of branches that are likely to be correlated with the NFA state but ignores branches that are unlikely to be correlated with NFA state. This makes it analysis more accurate, yet polynomial-time.

4.5 Summary of Algorithms

The "one value, one procedure, all paths" is the base version of the algorithm. The enhancements can be added individually and in any order. The initial plan for YASC is to add the enhancements in the order they are presented here.

5 Implementation and Current Status

The implementation will use SUIF as its infrastructure. SUIF includes a data-flow analysis framework and an alias analysis module.

The present prototype of YASC performs the "one value, one procedure, all paths" analysis.

6 Related Work

There are several temporal property verifiers. SLAM [BR00] verifies properties expressed as state machines in a C-like language called SLIC. SLAM does not interpret SLIC specifications as universally quantified. Instead, the specification writer uses something called the "universal quantification trick". SLAM works iteratively, first running on all paths. When it finds an apparent error, it checks the feasibility of the error path and repeats analysis if necessary.

BLAST, Berkeley Lazy Abstraction Software Verification Tool [HJMS02], is another verifier. BLAST is similar to SLAM, but it does much of the work lazily to improve performance.

ESP [DLS02] is the basis for YASC and is described above.

In the field of security, some intrusion-detection systems use similar NFAs and matching algorithms. The systems are based on automata that match attack scenarios. USTAT [IKP95] used pattern-matching to detect intrusion scenarios. STATL is a language for representing attack scenarios based on USTAT. IDIOT [KS94a, KS94b, KS95] implemented pattern-matching with unification using colored Petri nets.

Colored Petri nets are a formal language for design, specification, simulation, and verification of systems. A colored Petri net is an automaton that contains type information and names that can be bound to values. Many papers have been written on the theory and applications of colored Petri nets [Jen96a, KCJ98], including methods of constructing smaller equivalent automata [Jen96b].

Strauss [ABL02] is a specification miner. Specification miners infer property specifications from existing programs, which partially automates the writing of specifications.

7 References

[ABL02] Glenn Ammons, Ras Bodik, Jim Larus. Mining Specifications with Strauss. PLDI 2002.

[BR00] Thomas Ball, Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. Software Productivity Tools, Microsoft Research.

[DLS02] Manuvir Das, Sorin Lerner, Mark Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. PLDI, Berlin, 2002.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. In ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, Portland, Oregon, 2002.

[IKP95] Koral Ilgun, Richard Kemmerer, Phillip Porras. State Transition Analysis: A Rule-Based Intrusion Detection Approach. IEEE Transactions on Software Engineering, 1995.

[Jen96a] Kurt Jensen. An Introduction to the Practical Use of Colored Petri Nets. Department of Computer Science, University of Aarhus, Denmark, 1996.

[Jen96b] K. Jensen. Condensed state spaces for symmetrical coloured Petri nets. In Formal Methods in System Design.vol. 9, Kluwer Academic Publishers, 1996.

[KCJ98] L.M. Kristensen, S. Christensen, K. Jensen: The Practitioner's Guide to Coloured Petri Nets. International Journal on Software Tools for Technology Transfer, 2 (1998), Springer Verlag, 98-132.

[KS94a] Sandeep Kumar, Eugene Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report CSD-TR-94-103, Department of Computer Sciences, Purdue University, West Lafayette, IN, June 1994.

[KS94b] Sandeep Kumar, Eugene Spafford. A Pattern-Matching Model for Misuse Intrusion Detection. Proceedings of the 17th National Computer Security Conference, 1994.

[KS95] Sandeep Kumar, Eugene Spafford. A Software Architecture to Support Misuse Intrusion Detection. 1995.