Path Sensitive Program Analysis Using Boolean Satisfiability

Yichen Xie
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.
yxie@cs.stanford.edu

Andy Chou
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.
acc@cs.stanford.edu

ABSTRACT

Recent advances in boolean satisfiability (SAT) solvers have made it possible to solve structured formulas on the order of a million variables. In this work we show how to efficiently transform program analysis problems directly into SAT instances. The translation is similar to generating verification conditions, but we avoid exponential growth by introducing temporary variables in place of using substitution. Unlike most approaches based on general purpose theorem provers, our transformation deals with modular arithmetic and bit operations in a natural and efficient manner.

Our goal is to use SAT to solve some of the more difficult problems in applying dataflow techniques to automatically finding bugs in programs. We separate this task into two parts. First, an input program in a conventional programming language is filtered through a program abstractor. The output of the abstractor is a program in a simplified imperative programming language, with some properties annotated with assertions. Next, we translate this simplified language into SAT to check these assertions. We apply this technique to checking state machine properties, determining path feasibility, and detecting buffer overruns.

1. MOTIVATION

Static analysis has been shown to be very effective at finding errors in large systems. A typical approach is to specify an invariant property of an interface as an abstract state machine. Once a property is specified, a program abstractor determines the (possibly null) effect of each statement on the abstract state machine. This formulation leads naturally to an inter-procedural dataflow analysis which determines the set of reachable states for each program point. The tool reports an error for each program point that can be reached in an error state.

One problem with this approach is that false paths caused by correlated branches can lead to spurious error reports. In our approach, we take a demand-driven approach to this problem. First, a relatively cheap dataflow analysis like that performed by MC [14] or ESP [9] is performed to determine if any errors are possible. If no errors are possible, no further steps are necessary. If an error is reported, we translate the statements and control flow of the offending procedure into a boolean satisfiability (SAT) formula that is satisfiable if the error is reachable. Recent advances in SAT solvers have made it possible to solve instances with millions of variables [18, 22], and our preliminary results show that instances derived from small to moderate sized programs can be handled by these solvers.

One benefit of this approach is flexibility. It will be sound if the effect of each statement is modeled conservatively by the program abstractor. On the other hand, a sound approach may require very conservative assumptions, resulting in too many false error reports for certain programs or properties. This approach allows us to trade off soundness for usefulness. For example, the program abstractor can take advantage of ad-hoc conventions or domain-specific knowledge to avoid spurious error reports in a potentially unsound way.

In our basic approach, each value is precisely modeled at the bit level by a SAT formula, so the analysis can handle low-level features such as modular arithmetic and bitwise operations. Thus, we believe that our approach will be applicable to detecting integer overflows, off-by-one errors, and potentially buffer overflows, in addition to finite state machine properties.

2. TRANSLATION RULES

In this section, we describe the translation of programs from a small imperative language called Simple into boolean formulas. The translation outlined in this section is similar to the verification condition generation algorithm given by Flanagan and Saxe [13], except that the result is a boolean formula instead of a formula containing more complex operations on variables.

A boolean formula f is described by the grammar

$$f ::= v \mid \neg f \mid f \wedge f \mid f \vee f$$

where Vars is a set of boolean variables such that $v \in \mathsf{Vars}$. Boolean operators such as $\rightarrow$, $=$, and $\oplus$ (xor) can be reduced to the basic operators, and we will use them freely when building boolean formulas.

The translation from Simple to boolean formulas is done in two steps. In the first step, we simplify the source program by conservatively transforming while loops into if statements,

$$
\begin{array}{lcl}
S & ::= & \mathsf{skip} \\
 & | & S; S \\
 & | & V \leftarrow E \\
 & | & \mathsf{if}\ (B)\ \ S\ \ \mathsf{else}\ \ S \\
 & | & \mathsf{while}(B)\ S \\
 & | & \mathsf{assume}(B) \\
B & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg B \mid B \wedge B \mid B \vee B \mid E\ \mathsf{comp}\ E \\
E & ::= & V \mid C \mid \mathsf{unknown} \mid \mathsf{uop}\ E \mid E\ \mathsf{binop}\ E
\end{array}
$$

Figure 1: Abstract syntax for Simple, small imperative language. V represents variables, and C represents constants. In this paper uop $\in \{-\}$, binop $\in \{+, -,$ band $\}$ where band is bitwise and, and comp $\in \{<, =, \neq\}$.

and renaming variables so that each variable is only assigned once along any path. This results in a simpler program in a simpler language which we call, unsurprisingly, Simpler. In the second phase, we translate the simplified program into boolean formulas.

2.1 Source Language

Figure 1 gives the definition of Simple, the source language of the translation. Simple is a small imperative language with assignments, if statements, and while loops. Expressions are side-effect free; we handle a variety of operators ranging from modular addition and subtraction to bitwise and, or, and negation. The type system is trivial: values are either boolean (bool) or n-bit integers (int). All variables have type int. The expression unknown is a nondeterministic int value, which is used to model loops and unsupported operations. We use $\{\cdots\}$ to group statements and $(\cdots)$ to group expressions when it helps clarity.

2.2 Phase I: Handling Assignments and Loops

In an imperative language like C, variables can be and usually are assigned multiple times during their lifespan. Therefore they can take on different values at different program points. But in a boolean formula, each boolean variable can only take on one value. To bridge this gap, we borrow an idea from the SSA transformation [8]: we rename variables so that they are assigned only once on each path and therefore can only take on one value on each path.

In order to rename variables at assignments, we use a symbol table. The symbol table maps variables in the program to fresh variables not in the original program. The symbol table operations we use are:

- $T(v)$ looks up the variable v in the symbol table T and returns the result.
- $\mathsf{Rename}(T, V)$ takes a table T and a set of variables V from the original program and returns a new symbol table with each $v \in V$ mapped to a fresh variable not used anywhere else, and each $v \notin V$ mapped to $T(v)$. If the set is empty, Rename returns the table unchanged.

Finally, we use the functions $\mathsf{Def}(S)$ and $\mathsf{Ref}(S)$ to denote the set of all variables defined and referenced in statement S and all of its sub-statements.

With these symbol table operations, we can express the rules for the first step of the translation, shown in Figure 2. The rules define three translation functions, one for each syntactic category: $\overset{\mathsf{E}}{\leadsto}$ for int expressions, $\overset{\mathsf{B}}{\leadsto}$ for bool expressions, and $\overset{\mathsf{S}}{\leadsto}$ for statements.

$$
\begin{array}{l}
\overset{\mathsf{E}}{\leadsto}: \mathsf{expr_i} \times \mathsf{table} \rightarrow \mathsf{expr_i} \\
\overset{\mathsf{B}}{\leadsto}: \mathsf{expr_b} \times \mathsf{table} \rightarrow \mathsf{expr_b} \\
\overset{\mathsf{S}}{\leadsto}: \mathsf{stmt} \times \mathsf{table} \rightarrow \mathsf{stmt} \times \mathsf{table}
\end{array}
$$

The function $\overset{\mathsf{E}}{\leadsto}$ takes an int expression and a table and recursively replaces each variable in the expression with the variable it maps to in the table. The function $\overset{\mathsf{B}}{\leadsto}$ does the same thing for bool expressions. The function $\overset{\mathsf{S}}{\leadsto}$ takes a statement and a table and results in a new statement where all variables assume only one value on any path and while loops are desugared. The function $\overset{\mathsf{S}}{\leadsto}$ also returns a new table with the current symbol table mapping at the end of the statement. Given a program P which consists of a single statement, we can derive the transformed program and a final table as:

$$\langle P, \mathsf{Rename}(\{\}, \mathsf{Ref}(P) \cup \mathsf{Def}(P))\rangle \overset{\mathsf{S}}{\leadsto} \langle P', T\rangle$$

where $\mathsf{Rename}(\{\}, \mathsf{Ref}(P) \cup \mathsf{Def}(P))$ is just a fresh table mapping every variable in the program into fresh variables.

The core rules for the first step of the transformation are Assign, Seq, If, and While. We illustrate how these rules work on a few examples in Figure 3. Figure 3a illustrates how the rule for Assign renames a variable. It also illustrates how the rule for If introduces variables to unify the value after taking either of the branches. The introduced variables encode the intuition that if the true branch is taken, then the variables after the join point will have values from the true branch, and vice versa for the false branch.

Figure 3b illustrates how the While rule conservatively models the effect of loops. We translate a loop as a conditional branch that either (1) executes the loop body one or more times if the entry condition is true, or (2) never executes if the entry condition is false. We simulate the effect of (1) by assigning all variables defined in the loop body to unknown, and then assuming the loop condition. This sets the variables to arbitrary values that could have caused entry into the loop body. After the loop body, we assume the negation of the loop condition, thus forcing the variables to arbitrary values that would force exit of the loop, thus simulating the last iteration of the loop. Without loop invariants, this approximation can be imprecise.

We use the functions exec for the dynamic semantics of Simple:

$$\mathsf{exec} : \mathsf{stmt} \times \mathsf{env} \rightarrow 2^{\mathsf{env}}$$

For lack of space we omit a formal definition of exec. The exec function takes statements in Simple and an initial environment mapping variables to values and returns a set of possible final environments after executing the statement. The function is undefined if the statement loops forever, therefore exec is a partial function. We assume that the initial environment gives initial values to all variables used in the program. The return value is a set of environments because exec is nondeterministic; the nondeterminism is introduced by the unknown expression, which is used to model the nondeterministic choices of values in desugared while loops. Also, we will use the notation $T[\eta]$ to mean $\eta \circ T^{-1}$. That is, if $\eta(v) = a$ and $T(v) = v_i$ then $T[\eta]$ maps $v_i \mapsto a$.

Statements:

$$\frac{}{\langle \mathsf{skip}, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle \mathsf{skip}, T\rangle}\,\mathsf{Skip}$$

$$\frac{\langle b, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} b'}{\langle \mathsf{assume}(b), T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle \mathsf{assume}(b'), T\rangle}\,\mathsf{Assume}$$

$$\frac{\langle S_1, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle S_1', T'\rangle,\ \langle S_2, T'\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle S_2', T''\rangle}{\langle S_1; S_2, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle S_1'; S_2', T''\rangle}\,\mathsf{Seq}$$

$$\frac{v'\ \mathsf{Fresh},\ \langle e, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} e'}{\langle v \leftarrow e, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle v' \leftarrow e', T[v \mapsto v']\rangle}\,\mathsf{Assign}$$

$$\frac{\begin{array}{l}\langle S_t, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle S_t', T_t\rangle,\ \langle S_f, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle S_f', T_f\rangle,\ \langle b, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} b' \\ D = \mathsf{Def}(S_t) \cup \mathsf{Def}(S_f) \\ T' = \mathsf{Rename}(T, D) \\ A_t = \prod_{v \in D} T'(v) \leftarrow T_t(v) \\ A_f = \prod_{v \in D} T'(v) \leftarrow T_f(v)\end{array}}{\langle \mathsf{if}\ (b)\ S_t\ \mathsf{else}\ S_f, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle \mathsf{if}\ (b')\ \{S_t'; A_t\}\ \mathsf{else}\ \{S_f'; A_f\}, T'\rangle}\,\mathsf{If}$$

$$\frac{\begin{array}{l}A = \prod_{v \in \mathsf{Def}(S)} (v \leftarrow \mathsf{unknown}) \\ G = \mathsf{if}\ (b)\ \{A; \mathsf{assume}(b); S; \mathsf{assume}(\neg b)\}\ \mathsf{else}\ \{\mathsf{skip}\} \\ \langle G, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle G', T'\rangle\end{array}}{\langle \mathsf{while}(b)\ S, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle G', T'\rangle}\,\mathsf{While}$$

bool Expressions:

$$\frac{}{\langle \mathsf{true}, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} \mathsf{true}}\,\mathsf{True} \qquad \frac{\langle b, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} b'}{\langle \neg b, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} \neg b'}\,\mathsf{Not}$$

$$\frac{\langle b_1, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} b_1',\ \langle b_2, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} b_2'}{\langle b_1 \wedge b_2, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} b_1' \wedge b_2'}\,\mathsf{And}$$

$$\frac{\langle e_1, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} e_1',\ \langle e_2, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} e_2'}{\langle e_1\ \mathsf{comp}\ e_2, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} e_1'\ \mathsf{comp}\ e_2'}\,\mathsf{Comp}$$

int Expressions

$$\frac{T(v) = v'}{\langle v, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} v'}\,\mathsf{Var} \qquad \frac{}{\langle c, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} c}\,\mathsf{Const}$$

$$\frac{}{\langle \mathsf{unknown}, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} \mathsf{unknown}}\,\mathsf{Unknown}$$

$$\frac{\langle e, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} e'}{\langle \mathsf{uop}\ e, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} \mathsf{uop}\ e'}\,\mathsf{Uop}$$

$$\frac{\langle e_1, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} e_1',\ \langle e_2, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} e_2'}{\langle e_1\ \mathsf{binop}\ e_2, T\rangle \overset{\mathtt{E}}{\rightsquigarrow} e_1'\ \mathsf{binop}\ e_2'}\,\mathsf{Binop}$$

Figure 2: Phase I of the translation: we remove assignments and desugar while loops. For a set $D = \{x, y, z\}$, the meaning of $\prod_{v \in D} v$ is $x; y; z$. Some straightforward rules are omitted.

$$\begin{array}{l}\mathsf{if}\ (c \neq 0) \\ \quad v \leftarrow v + 1 \\ \mathsf{else} \\ \quad v \leftarrow v - 1 \\ v \leftarrow v + 1\end{array} \quad \overset{\mathtt{S}}{\rightsquigarrow} \quad \begin{array}{l}\mathsf{if}\ (c_0 \neq 0)\ \{ \\ \quad v_1 \leftarrow v_0 + 1; \\ \quad v_3 \leftarrow v_1 \\ \}\ \mathsf{else}\ \{ \\ \quad v_2 \leftarrow v_0 - 1; \\ \quad v_3 \leftarrow v_2 \\ \} \\ v_4 \leftarrow v_3 + 1\end{array}$$

$\mathsf{Initial}: [c \mapsto c_0, v \mapsto v_0]$ $\mathsf{Final}: [c \mapsto c_0, v \mapsto v_4]$

(a)

$$\begin{array}{l}\mathsf{while}(v < 10) \\ \quad v \leftarrow v + 1\end{array} \quad \overset{\mathtt{S}}{\rightsquigarrow} \quad \begin{array}{l}\mathsf{if}\ (v_0 < 10)\ \{ \\ \quad v_1 \leftarrow \mathsf{unknown}; \\ \quad \mathsf{assume}(v_1 < 10); \\ \quad v_2 \leftarrow v_1 + 1; \\ \quad \mathsf{assume}(\neg(v_2 < 10)); \\ \quad v_3 \leftarrow v_2 \\ \}\ \mathsf{else}\ \{ \\ \quad v_3 \leftarrow v_0 \\ \}\end{array}$$

$\mathsf{Initial}: [v \mapsto v_0]$ $\mathsf{Final}: [v \mapsto v_3]$

(b)

Figure 3: Example translations of an if statement and a while statement with initial and final tables.

With that, we can state the soundness lemma for Phase I of the translation.

Lemma 1 (Phase I of the translation is sound) Given Simple program S, initial store η and a well formed symbol table T, if

$$\langle S, T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle S', T'\rangle,$$

then

$$T'[\mathsf{exec}(S, \eta)] \subseteq \mathsf{exec}(S', T[\eta])$$

Proof: by structural induction on translation rules in phase 1.

2.3 Phase II: Translating to Boolean Formulas

Figure 4 shows the rules for the second phase of the translation. Like the rules for the first phase, the rules for this phase specify three translation functions:

$$\begin{array}{l} \xrightarrow{\mathtt{E}}: \mathsf{expr_i} \rightarrow \mathsf{array(f)} \\ \xrightarrow{\mathtt{B}}: \mathsf{expr_b} \rightarrow \mathsf{f} \\ \xrightarrow{\mathtt{S}}: \mathsf{stmt} \rightarrow \mathsf{f} \end{array}$$

The $\xrightarrow{\mathtt{E}}$ function results in an array of boolean formulas, each representing the symbolic value of one bit of the result. The $\xrightarrow{\mathtt{B}}$ function results in a single boolean formula which is satisfiable if the boolean expression can evaluate to true. The $\xrightarrow{\mathtt{S}}$ function returns a boolean formula that has at least one satisfying assignment for every possible terminating execution of the statement. The translation uses the function $\mathsf{BitVec}(v)$ which maps every int variable v to an array of unique boolean variables.

For example, consider the expression $x = y$, which evaluates to true iff x and y have the same value, false otherwise. The rule Equal says that to translate $x = y$, first translate x into an array of formulas $[x_n \dots x_1]$ and similarly for y. Then the rule specifies the result of translating $x = y$ to be:

$$\bigwedge_{i=1}^{n} x_i = y_i$$

This boolean formula will evaluate to true if all bits x_i of x have the same value as the corresponding bits y_i of y, and false otherwise.

Likewise, addition and subtraction between variables translates into a boolean formula that simulates a ripple-carry adder. Because we use 2's complement representation for integers, addition and subtraction automatically treat arithmetic overflow in the same way as a real machine does. Comparison operators can be translated directly by comparing each bit in turn. Bit-wise operations are trivial to translate because the representation keeps a formula for each bit. We use a variant of Booth's algorithm to translate multiplication between variables and constants, but we do not support multiplication between variables. In general, it is sound but imprecise to model the value of unsupported operators with an array of fresh boolean variables, which essentially says that the expression can return any value in its type.

Lemma 2 states that the second phase of the translation is sound. We use $\mathsf{BitStore}(\eta)$ to convert the environment η into its bit level counterpart. More precisely, if $\mathsf{BitVec}(v) = [v_n \dots v_1]$ and $\eta(v) = a$, then $\mathsf{BitStore}(\eta)$ maps $v_i \mapsto ((a \;\mathsf{band}\; 2^{i-1}) \neq 0)$.

$\boxed{\text{Statements:}}$

$$\frac{}{\mathsf{skip} \xrightarrow{\mathtt{S}} \mathsf{true}}\,\mathsf{Skip} \qquad \frac{(v = e) \xrightarrow{\mathtt{B}} f}{(v \leftarrow e) \xrightarrow{\mathtt{S}} f}\,\mathsf{Assign}$$

$$\frac{S_1 \xrightarrow{\mathtt{S}} f_1,\; S_2 \xrightarrow{\mathtt{S}} f_2}{S_1; S_2 \xrightarrow{\mathtt{S}} f_1 \wedge f_2}\,\mathsf{Seq}$$

$$\frac{b \xrightarrow{\mathtt{B}} f_b,\; S_t \xrightarrow{\mathtt{S}} f_t,\; S_f \xrightarrow{\mathtt{S}} f_f}{\mathsf{if}\ (b)\ S_t\ \mathsf{else}\ S_f \xrightarrow{\mathtt{S}} (f_b \wedge f_t) \vee (\neg f_b \wedge f_f)}\,\mathsf{If}$$

$$\frac{b \xrightarrow{\mathtt{B}} f}{\mathsf{assume}(b) \xrightarrow{\mathtt{S}} f}\,\mathsf{Assume}$$

$\boxed{\text{bool Expressions:}}$

$$\frac{}{\mathsf{true} \xrightarrow{\mathtt{B}} \mathsf{true}}\,\mathsf{True} \qquad \frac{}{\mathsf{false} \xrightarrow{\mathtt{B}} \mathsf{false}}\,\mathsf{False}$$

$$\frac{b \xrightarrow{\mathtt{B}} f}{\neg b \xrightarrow{\mathtt{B}} \neg f}\,\mathsf{Not} \qquad \frac{b_1 \xrightarrow{\mathtt{B}} f_1,\; b_2 \xrightarrow{\mathtt{B}} f_2}{b_1 \wedge b_2 \xrightarrow{\mathtt{B}} f_1 \wedge f_2}\,\mathsf{And}$$

$$\frac{e_1 \xrightarrow{\mathtt{E}} [f_n \dots f_1],\; e_2 \xrightarrow{\mathtt{E}} [g_n \dots g_1]}{e_1 = e_2 \xrightarrow{\mathtt{B}} \bigwedge_{i=1}^{n} f_i = g_i}\,\mathsf{Equal}$$

$$\frac{e_1 \xrightarrow{\mathtt{E}} [f_n \dots g_1],\; e_2 \xrightarrow{\mathtt{E}} [g_n \dots g_1]}{e_1 < e_2 \xrightarrow{\mathtt{B}} (f_n \wedge \neg g_n) \vee (f_n = g_n \wedge (\dots))}\,\mathsf{Less}$$

$\boxed{\text{int Expressions:}}$

$$\frac{\mathsf{BitVec}(v) = [v_n \dots v_1]}{v \xrightarrow{\mathtt{E}} [v_n \dots v_1]}\,\mathsf{Var}$$

$$\frac{v_1, \dots, v_n\ \mathsf{Fresh}}{\mathsf{unknown} \xrightarrow{\mathtt{E}} [v_n \dots v_1]}\,\mathsf{Unknown}$$

$$\frac{e_1 \xrightarrow{\mathtt{E}} [f_n \dots f_1],\; e_2 \xrightarrow{\mathtt{E}} [g_n \dots g_1]}{e_1\ \mathsf{band}\ e_2 \xrightarrow{\mathtt{E}} [f_n \wedge g_n; \dots ; f_1 \wedge g_1]}\,\mathsf{BitwiseAnd}$$

$$\frac{e_1 \xrightarrow{\mathtt{E}} [f_n \dots f_1],\; e_2 \xrightarrow{\mathtt{E}} [g_n \dots g_1]}{e_1 + e_2 \xrightarrow{\mathtt{E}} [\dots ; f_2 \oplus g_2 \oplus (f_1 \wedge g_1); f_1 \oplus g_1]}\,\mathsf{Plus}$$

Figure 4: Second phase of the translation. These rules convert control flow and expressions into boolean formulas, and they assume that there are no while loops in the input. The Assign rule takes advantage of the renaming done in the first phase and simply becomes an equality assumption. The Less rule implements an unsigned comparison; a signed comparison operator is only slightly more complex.

Lemma 2 (Phase II of the translation is sound) Given $\mathsf{Simpler}$ program S from phase I, let η be the initial state, and η' be the result of one possible execution, i.e.

$$\eta' \in \mathsf{exec}(S, \eta)$$

If

$$S \xrightarrow{\mathtt{S}} f$$

Then $\mathsf{BitStore}(\eta')$ is a model for f.

Proof: by structural induction on translation rules in phase II.

2.4 Putting it all together

With Phase I and II of the translation defined above, we define function $\mathsf{SimpleToSAT}$ that puts these phases together:

$$\frac{\langle P, \mathsf{Rename}(\{\}, \mathsf{Ref}(P) \cup \mathsf{Def}(P))\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle P', T\rangle \quad P' \xrightarrow{\mathtt{S}} f}{\mathsf{SimpleToSAT}(P') = f}\,\mathsf{SimpleToSAT}$$

Theorem 1 ($\mathsf{SimpleToSAT}$ is sound) Given P, if

$$\mathsf{SimpleToSAT}(P) = f$$

Then a valid execution of P will correspond to a model for f.

Proof: by Lemma 1 and 2. Note the correspondence is embedded in the application of the lemmas.

3. STATIC ASSERTION CHECKER

One natural application of the translation framework is a static assertion checker. The source language used in this section will be Simple augmented with assertions:

$$S ::= \ldots \mid \mathsf{assert}(B)$$

We call this augmented language $\mathsf{Simple}^{\mathsf{A}}$. The `assert()` facility is found in many programming languages. It allows programmers to abort execution immediately if the boolean argument is not true. Robust code uses assertions extensively as a form of documentation and reassurance. Automated instrumentation tools such as SafeC [2], CCured [19], and others also introduce assertions that help enforce safety properties that the user forgets to check.

We use Phase I of the translation (Figure 2) to turn $\mathsf{Simple}^{\mathsf{A}}$ into $\mathsf{Simpler}^{\mathsf{A}}$ (i.e. $\mathsf{Simple}^{\mathsf{A}}$ with renamed variables and desugared while loops). The only change necessary to handle assert is one additional rule, shown in Part 1 of Figure 5. The rule renames the variables in the assert predicate using the given table.

Part 2 of Figure 5 defines a new function:

$$\xrightarrow{\mathtt{A}}: \mathsf{stmt} \to \mathsf{f}$$

which takes a statement from $\mathsf{Simpler}^{\mathsf{A}}$ and returns a boolean formula which is satisfiable if the statement has a potential assertion failure. The intuition for $\xrightarrow{\mathtt{A}}$ is that assignments, assumptions, and skip can never cause assertion failure by themselves. A sequence can cause an error if an error occurs in the first statement, or if the first statement executes normally and the second statement causes an error. An if statement can cause an error when the condition is true and the true branch causes an error, and similarly for the false branch. Finally, an assertion statement can fail if the condition is false.

The soundness of $\xrightarrow{\mathtt{A}}$ follows from Theorem 1 and a structural induction on the rules in Figure 5.

$\boxed{\text{Part 1:}}$

$$\frac{\langle b, T\rangle \overset{\mathtt{B}}{\rightsquigarrow} b'}{\langle \mathsf{assert}(b), T\rangle \overset{\mathtt{S}}{\rightsquigarrow} \langle \mathsf{assert}(b'), T\rangle}\,\mathsf{Assert}$$

$\boxed{\text{Part 2:}}$

$$\frac{}{\mathsf{skip} \xrightarrow{\mathtt{A}} \mathsf{false}}\,\mathsf{Skip} \qquad \frac{}{(v \leftarrow e) \xrightarrow{\mathtt{A}} \mathsf{false}}\,\mathsf{Assign}$$

$$\frac{S_1 \xrightarrow{\mathtt{S}} g,\ S_1 \xrightarrow{\mathtt{A}} f_1,\ S_2 \xrightarrow{\mathtt{A}} f_2}{S_1; S_2 \xrightarrow{\mathtt{A}} f_1 \vee (g \wedge f_2)}\,\mathsf{Seq}$$

$$\frac{b \xrightarrow{\mathtt{B}} f_b,\ S_t \xrightarrow{\mathtt{A}} f_t,\ S_f \xrightarrow{\mathtt{A}} f_f}{\mathsf{if}\ (b)\ S_t\ \mathsf{else}\ S_f \xrightarrow{\mathtt{A}} (f_b \wedge f_t) \vee (\neg f_b \wedge f_f)}\,\mathsf{If}$$

$$\frac{}{\mathsf{assume}(b) \xrightarrow{\mathtt{A}} \mathsf{false}}\,\mathsf{Assume}$$

$$\frac{b \xrightarrow{\mathtt{B}} f}{\mathsf{assert}(b) \xrightarrow{\mathtt{A}} \neg f}\,\mathsf{Assert}$$

Figure 5: Rules for the assertion checking translation.

4. STATE MACHINE PROPERTIES

One large class of temporal safety properties can be expressed in terms of finite state machines (FSM). This approach has been successful in finding a large number of errors in programs [12, 1]. One limitation of this approach is the difficulty of adding path sensitivity. For most properties, it is hard for path-insensitive tools to go beyond intra-procedural analysis without flagging a large number of false positives. In this section we propose an FSM analyzer based on the Simple to SAT translation framework. We present two different FSM checkers: one is a direct extension to the assertion checker in Section 3 suitable for intra-procedural analysis; a more complex version employs program slicing techniques and may be suitable for verifying whole programs.

The source language in this section will be Simple augmented with state transitions:

$$S ::= \ldots \mid [s_1 \Rightarrow s_1'; \ldots; s_n \Rightarrow s_n']$$

where the distinct states s_i are drawn from a finite set of states $\mathcal{D}$ with a distinguished state $\mathsf{error} \in \mathcal{D}$. In this paper we will assume for simplicity that $\mathcal{D} \subseteq \mathsf{int}$, but we will use names to refer to states. We assume the initial state is given in the execution environment. We call this augmented language $\mathsf{Simple}^{\mathsf{T}}$.

4.1 Simple FSM Checker

Merging Transitions

$$\frac{}{\{\} \bigoplus_S \{\} = \{\}} \qquad \frac{}{\{\} \bigoplus_S \{\omega\} = \{\omega\}} \qquad \frac{}{\{\omega\} \bigoplus_S \{\} = \{\omega\}}$$

$$\frac{\begin{array}{l} \omega_1 = \langle s_1 \Rightarrow s_2, g_1, V_1 \rangle \\ \omega_2 = \langle s_1 \Rightarrow s_2, g_2, V_2 \rangle \\ \langle f, \_ \rangle = \mathsf{SliceTrans}(S, V_1 \cup V_2) \\ g = \begin{cases} \mathsf{true} & \text{if } (f \rightarrow g_1 \vee g_2) \text{ is tautology} \\ g_1 \vee g_2 & \text{otherwise} \end{cases} \\ V = \begin{cases} \{\} & \text{if } g = \mathsf{true} \\ V_1 \cup V_2 & \text{otherwise} \end{cases} \end{array}}{\{\omega_1\} \bigoplus_S \{\omega_2\} = \langle s_1 \Rightarrow s_2, g, V \rangle}$$

$$\begin{array}{l} \mathsf{Merge}_S(\Omega_A, \Omega_B) = \\ \quad \bigcup_{s_1, s_2 \in \mathcal{D}} \left(\mathsf{Proj}_{s_1,s_2}(\Omega_A)\right) \bigoplus_S \left(\mathsf{Proj}_{s_1,s_2}(\Omega_B)\right) \\ \mathsf{MergeSet}_S(\{\Omega_1, \ldots, \Omega_n\}) = \\ \quad \bigcup_{s_1, s_2 \in \mathcal{D}} \left(\left(\mathsf{Proj}_{s_1,s_2}(\Omega_1)\right) \bigoplus_S \ldots \bigoplus_S \left(\mathsf{Proj}_{s_1,s_2}(\Omega_n)\right)\right) \end{array}$$

Composing Transitions

$$\frac{\begin{array}{c} \omega_1 = \langle s_1 \Rightarrow s_2, g_1, V_1 \rangle \\ \omega_2 = \langle s_2 \Rightarrow s_3, g_2, V_2 \rangle \\ \langle f, \_ \rangle = \mathsf{SliceTrans}(S, V_1 \cup V_2) \\ f \wedge (g_1 \wedge g_2) \text{ satisfiable} \end{array}}{\omega_1 \bigotimes_S \omega_2 = \{\langle s_1 \Rightarrow s_3, g_1 \wedge g_2, V_1 \cup V_2 \rangle\}}$$

$$\frac{\begin{array}{c} \omega_1 = \langle s_1 \Rightarrow s_1', g_1, V_1 \rangle \\ \omega_2 = \langle s_2 \Rightarrow s_2', g_2, V_2 \rangle \\ \langle f, \_ \rangle = \mathsf{SliceTrans}(S, V_1 \cup V_2) \\ s_1' \neq s_2 \text{ or } (f \wedge g_1 \wedge g_2) \text{ not satisfiable} \end{array}}{\omega_1 \bigotimes_S \omega_2 = \{\}}$$

$$\frac{\begin{array}{c} \Omega_A = \{\omega_{A1}, \ldots, \omega_{Am}\} \\ \Omega_B = \{\omega_{B1}, \ldots, \omega_{Bn}\} \end{array}}{\begin{array}{l} \mathsf{Compose}_S(\Omega_A, \Omega_B) = \\ \quad \mathsf{MergeSet}_S(\bigcup_{i,j} (\omega_{Ai} \bigotimes_S \omega_{Bj})) \end{array}} \text{Compose}$$

Figure 8: Composing and merging transitions. The function $\mathsf{Proj}_{s_1,s_2}(\Omega)$ is the projection of the set Ω to the (always empty or singleton) set of triples which have a transition of $s_1 \Rightarrow s_2$.

$$\frac{}{\mathsf{SliceTrans}(\mathsf{skip}, V) = \langle \mathsf{true}, V \rangle} \text{Slice-Skip}$$

$$\frac{}{\mathsf{SliceTrans}([s_1 \Rightarrow s_1'; \ldots ; s_n \Rightarrow s_n'], V) = \langle \mathsf{true}, V \rangle} \text{Slice-Trans}$$

$$\frac{\begin{array}{c} \langle f_2, V' \rangle = \mathsf{SliceTrans}(S_2, V) \\ \langle f_1, V'' \rangle = \mathsf{SliceTrans}(S_1, V') \end{array}}{\mathsf{SliceTrans}(S_1; S_2, V) = \langle f_1 \wedge f_2, V'' \rangle} \text{Slice-Seq}$$

$$\frac{(v = e) \xrightarrow{\mathsf{B}} f, \; v \in V}{\mathsf{SliceTrans}(v \leftarrow e, V) = \langle f, V \cup \mathsf{Ref}(e) \rangle} \text{Slice-Assign-Relevant}$$

$$\frac{v \notin V}{\mathsf{SliceTrans}(v \leftarrow e, V) = \langle \mathsf{true}, V \rangle} \text{Slice-Assign-Irrelevant}$$

$$\frac{\begin{array}{l} b \xrightarrow{\mathsf{B}} f_b \\ \langle f_t, V_t \rangle = \mathsf{SliceTrans}(S_t, V) \\ \langle f_f, V_f \rangle = \mathsf{SliceTrans}(S_f, V) \\ D = \mathsf{Def}(S_t) \cup \mathsf{Def}(S_f) \\ V' = \begin{cases} \mathsf{Ref}(b) & \text{if } D \cap V \neq \phi \\ \phi & \text{otherwise} \end{cases} \end{array}}{\begin{array}{l} \mathsf{SliceTrans}(\mathsf{if}\ (b)\ S_t\ \mathsf{else}\ S_f, V) = \\ \quad \langle (f_b \wedge f_t) \vee (\neg f_b \wedge f_f), V \cup V_t \cup V_f \cup V' \rangle \end{array}} \text{Slice-If}$$

Figure 6: Program Slicing

The simple way to encode state machine properties is to convert $\mathsf{Simple}^{\mathsf{T}}$ into $\mathsf{Simple}^{\mathsf{A}}$ by adding a special *state variable* to keep track of the current state of the FSM at each program point. FSM transitions are translated into conditional assignments to the state variable, followed by an assertion that the new state is not error. More precisely, the state machine transition:

$$[s_1 \Rightarrow s_1'; \ldots ; s_n \Rightarrow s_n']$$

Adding Guards to Transitions

$$\frac{\begin{array}{l} \Omega = \bigcup_{i=1}^{n} \{\langle s_i \Rightarrow s_i', g_i, V_i \rangle\} \\ c \xrightarrow{\mathsf{B}} c' \end{array}}{\begin{array}{l} \mathsf{AddGuard}(c, \Omega) = \\ \bigcup_{i=1}^{n} \{\langle s_i \Rightarrow s_i', c' \wedge g_i, V_i \cup \mathsf{Ref}(c) \rangle\} \end{array}} \text{AddCond}$$

FSM Summary Construction

$$\frac{S = \{v \leftarrow e\}}{S \xrightarrow{\mathsf{SM}} \bigcup_{s \in \mathcal{D}} \{\langle s \Rightarrow s, \mathsf{true}, \{\} \rangle\}} \text{Assign}$$

$$\frac{S = \mathsf{assume}(b)}{S \xrightarrow{\mathsf{SM}} \bigcup_{s \in \mathcal{D}} \{\langle s \Rightarrow s, \mathsf{true}, \{\} \rangle\}} \text{Assume}$$

$$\frac{S = [s_1 \Rightarrow s_1'; \ldots ; s_n \Rightarrow s_n']}{S \xrightarrow{\mathsf{SM}} \bigcup_{i=1}^{n} \{\langle s_i \Rightarrow s_i', \mathsf{true}, \{\} \rangle\}} \text{Transition}$$

$$\frac{\begin{array}{c} S = \{\ \mathsf{if}\ (b)\ \ S_t\ \ \mathsf{else}\ \ S_f\ \} \\ S_t \xrightarrow{\mathsf{SM}} \Omega_t \quad \Omega_t' = \mathsf{AddGuard}(b, \Omega_t) \\ S_f \xrightarrow{\mathsf{SM}} \Omega_f \quad \Omega_f' = \mathsf{AddGuard}(\neg b, \Omega_f) \end{array}}{S \xrightarrow{\mathsf{SM}} \mathsf{Merge}_S(\Omega_t', \Omega_f')} \text{If}$$

$$\frac{\begin{array}{c} S = S_1; S_2 \\ S_1 \xrightarrow{\mathsf{SM}} \Omega_1 \quad S_2 \xrightarrow{\mathsf{SM}} \Omega_2 \end{array}}{S_1; S_2 \xrightarrow{\mathsf{SM}} \mathsf{Compose}_S(\Omega_1, \Omega_2)} \text{Seq}$$

Figure 7: Constructing FSM Summaries.

in $\mathsf{Simple}^{\mathsf{T}}$ will be translated into the $\mathsf{Simple}^{\mathsf{A}}$ statement:

$$\begin{array}{l} \mathsf{if}\ (\mathsf{state} = s_1)\ \mathsf{state} \leftarrow s_1' \\ \mathsf{else\ if}\ (\mathsf{state} = s_2)\ \mathsf{state} \leftarrow s_2' \\ \quad \ldots \\ \mathsf{else\ if}\ (\mathsf{state} = s_n)\ \mathsf{state} \leftarrow s_n' \\ \mathsf{else\ assume(false)}; \\ \mathsf{assert}(\neg(\mathsf{state} = \mathsf{error})) \end{array}$$

The state variable is initialized to the start state on entering the procedure, i.e. given a program P and an initial state Init, we actually translate the program $P' = (\mathsf{state} \leftarrow \mathsf{Init}; P)$.

We check the resulting $\mathsf{Simple}^{\mathsf{A}}$ program with the algorithm shown in Section 3.

4.2 FSM Checker with Slicing

One potential problem with the simple FSM translation above is scalability. The simple translation takes into account all the program variables that may or may not be related to the FSM property we are checking. Consider the following example, which uses a state machine property involving locking and unlocking:

```
/* enter in the unlocked state */
if (x ≠ 0)
   [unlocked ⇒ locked; locked ⇒ error]
else skip;
y = /* complex computation */;
if (x ≠ 0)
   [locked ⇒ unlocked; unlocked ⇒ error]
else skip
```

In this example, the variable y is most likely not related to the lock/unlock parity property we check, and therefore can be safely omitted in the translation. Also, no matter what value x holds, the program segment will always exit in the unlocked state, so we do not have to track the value of x outside of this program segment.

In this section, we propose a solution that employs program slicing techniques [21]. The algorithm takes a $\mathsf{Simpler}^{\mathsf{T}}$ program P and computes a *summary* of P as a set of triples:

$$P \overset{\mathtt{SM}}{\rightarrow} \bigcup_{i=0}^{n} \{\langle s_i \Rightarrow s'_i, g_i, V_i \rangle\}$$

Each triple $\langle s \Rightarrow s', g, V\rangle$ in the summary describes one possible FSM transition ($s \Rightarrow s'$) through the program, annotated with a boolean formula g that is satisfiable if the transition can take place. We call g the *guard* of the transition. We also include a set of variables V that are referenced when constructing the guard g. V is later used as the slicing criterion when we simplify boolean formulas for testing path feasibility.

The rest of this section describes the rules for the slicing translation. Section 4.3 shows an example that illustrates how they work. The reader may wish to skip to that section first in order to build some intuition.

We use the slicer function $\mathsf{SliceTrans}(S, V)$ to remove parts of the statement S unrelated to the set of variables V (also called a slicing criterion). Rules for slicing are given in Figure 6. SliceTrans replaces Phase II of the SimpleToSAT translation, and removes the effect of assignments of irrelevant variables by replacing them with the effect of skip, which is true.

Figure 7 shows the algorithm that computes FSM summaries from $\mathsf{Simpler}^{\mathsf{T}}$ programs. The algorithm recursively computes a set of triples for each statement. These triples summarize the possible state transitions that occur within the statement. For assignment and assume statements, no state transition occurs, and the rule specifies that the result is merely a set of identity triples. That is, for every state $s \in \mathcal{D}$, each triple maps $s \Rightarrow s$, with a guard that is true and an empty variable set.

For transitions, the algorithm returns the set of triples with the transitions specified by the statement, with true guards and empty variable sets. This is because a transition statement unconditionally changes the state. If no transition is specified for a particular state, the algorithm assumes that the code path is not reachable in that state (this is the same as the simple FSM checker in Section 4.1).

For if statements, we add the condition to the guards of the triples derived from each branch, and then we merge the triples from each branch into a single summary. The AddGuard inserts the control predicate into the triples from the true and false branches, and the Merge function combines the two sets of triples into one. As shown in Figure 8, Merge merges triples with common transitions by or'ing their guards together. If the program execution entails the resulting guard (i.e. the condition $f \rightarrow g_1 \vee g_2$ is a tautology), then we know that the transition $s_1 \Rightarrow s_2$ always occurs, no matter what value variables have before the beginning of the if statement. In this case, we can ignore dependencies on variables before this point by replacing the guard with true, and clearing the relevant variable set. Note that testing whether $f \rightarrow g_1 \vee g_2$ is a tautology is optional because it is sound to always keep the guards and variable set.

To compute the summary triples for $\{S_A; S_B\}$, the Seq rule uses the Compose function to combine the transition summaries of S_A and S_B. As shown in Figure 8, Compose essentially chains the FSM transitions in its arguments and tests feasibility. A transition is only accepted into the final summary if it arises from a valid path. This feasibility test need not be done every time Compose is called; it can be done at any time. For example, the test can be done at the end of the program when all of the final triples have been determined. This way a single call to SAT will determine if there is a feasible path starting in a particular initial state and ending in the error state. If procedure calls are added to Simple, it would also make sense to determine path feasibility immediately before each function call; if the path is not feasible, the call need not be analyzed along that path.

Recall from Section 2 that a valid execution of a Simple program corresponds to a satisfiable boolean formula f from the SimpleToSAT translation. Therefore, the feasibility of the transition can be tested by conjuncting the guards with f and testing satisfiability of the result using the SAT solver. The result from the chaining process is further merged to produce the final summary.

Last but not least, recall that we desugar while loops into a single if statement that simulates the following two possibilities: (a) the loop body is never executed (b) the loop body is executed one or more times in which case we simulate the final iteration of the loop. For a finite state machine with n states, unrolling the loop once is not enough, because some transitions might only take place when the loop body is executed multiple times. To solve this problem, we need to unroll the loop a maximum of n times in the worst case. We introduce a fresh set of variables each time we unroll the loop body to simulate the effect that an arbitrary number of iterations have elapsed between each transition.

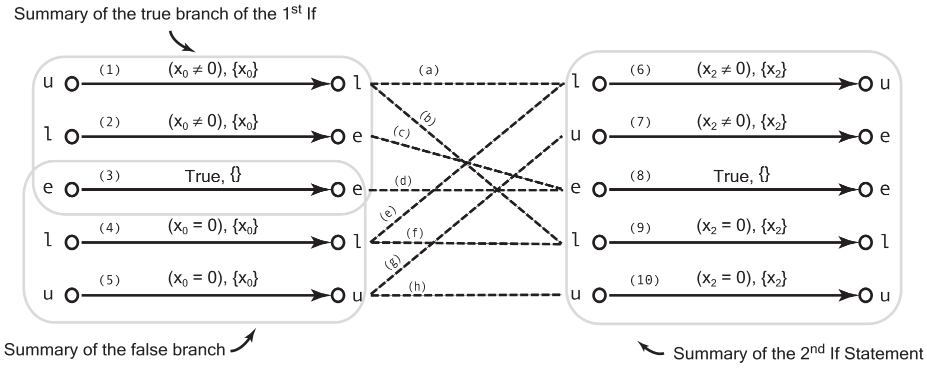

Figure 9: Example of the slicing translation.

```
if (x0 < 10) {
  /* +++ */
  assume(x1 < 10);
  x2 ← x1 + 1;
  if (x2 < 10) {
    /* +++ */
    assume(x3 < 10);
    x4 ← x3 + 1;
    /* unroll again here if necessary */
    x5 ← x4
  } else x5 ← x2;
  assume(¬(x5 < 10));
  x6 ← x5
} else x6 ← x0
```

Figure 10: Example showing loop unrolling for FSMs.

Because of space limitations, instead of writing out the specific rules, we use the following example to illustrate the translation:

```
while (x < 10)
   x <- x + 1;
```

The result of unrolling this loop twice is shown in Figure 10. Comments `/* +++ */` indicate the positions where an arbitrary number of iterations could have happened. In the implementation, we do not always need to unroll n times to obtain all possible transitions. An optimization would be to unroll until a fixed point is reached on the set of transitions.

4.3 Slicing Example

We will use the following example to show the translation rules in action. Suppose that after Phase I, we arrive at the following Simpler$^{\mathsf{T}}$ program:

```
1:  /* enter in the unlocked state */
2:  if (x0 ≠ 0) {
3:     [unlocked ⇒ locked; locked ⇒ error];
4:     x1 ← 2; x2 ← x1
5:  } else  x2 ← x0;
6:  if (x2 ≠ 0)
7:     [locked ⇒ unlocked; unlocked ⇒ error]
8:  else skip
```

Figure 9 illustrates the translation from Simpler$^{\mathsf{T}}$ to FSM summaries. Each triple $\langle s \Rightarrow s', g, V\rangle$ in the summary is represented by a solid arrow in the graph, going from the source state to the target state. The guard g and relevant variable set V are labeled on top of the arrow. Ignoring the labels, triples 1-3 are generated from the lock transition on line 3 by the Transition rule. Triples 3-5 are from the assignment on line 5. To compute the summary of the first if statement, we combine the set of triples from the two branches by applying the If rule. The effect of this merge is that the (error $\Rightarrow$ error) transition common to both branches is merged with the $\bigoplus$ operation. The result is shown as arrows 1-5, with the labels added by the AddCond operation in the If rule. Similarly, the second if statement computes to a summary represented by arrows 6-10.

To compute the summary for the whole program, we compose the two sets of transitions from both if statements using the Seq rule. Dotted lines (a)-(h) in the graph corresponds to the $\bigotimes$ operations applied in the process, each representing a possible chain of transitions, which we list below:

$$
\begin{array}{l}
\surd\ \texttt{(a) u} \xrightarrow{(x_0\neq0)\wedge(x_2\neq0),\ \{x_0,x_2\}} \texttt{u}\\
\times\ \texttt{(b) u} \xrightarrow{(x_0\neq0)\wedge(x_2=0),\ \{x_0,x_2\}} \texttt{l}\\
\surd\ \texttt{(c) l} \xrightarrow{(x_0\neq0)\wedge\mathsf{true},\ \{x_0\}} \texttt{e}\\
\surd\ \texttt{(d) e} \xrightarrow{\mathsf{true},\ \{\}} \texttt{e}\\
\times\ \texttt{(e) l} \xrightarrow{(x_0=0)\wedge(x_2\neq0),\ \{x_0,x_2\}} \texttt{u}\\
\surd\ \texttt{(f) l} \xrightarrow{(x_0=0)\wedge(x_2=0),\ \{x_0,x_2\}} \texttt{l}\\
\times\ \texttt{(g) u} \xrightarrow{(x_0=0)\wedge(x_2\neq0),\ \{x_0,x_2\}} \texttt{e}\\
\surd\ \texttt{(h) u} \xrightarrow{(x_0=0)\wedge(x_2=0),\ \{x_0,x_2\}} \texttt{u}
\end{array}
$$

For the variable set $V = \{x_0, x_2\}$, we slice the entire program P with SliceTrans(P, V) and get the following formula f:

$$(x_0 \neq 0 \wedge x_1 = 2 \wedge x_2 = x_1) \vee (x_0 = 0 \wedge x_2 = x_0)$$

Transitions (b), (e), and (g) are eliminated in this step because they arise from infeasible paths. We detect this because $f \wedge g$ is unsatisfiable for the guards g of these transitions. Finally, the $\bigoplus$ operation is called to combine (a) and (h), both of which goes from unlocked to unlocked. Since

$$f \rightarrow ((x_0 \neq 0) \wedge (x_2 \neq 0)) \vee ((x_0 = 0) \wedge (x_2 = 0))$$

is a tautology, we set the guard to be true, and arrive at the final result (i).

$$\texttt{(i) u} \xrightarrow{\mathsf{true},\ \{\}} \texttt{u}$$

This removes the dependence on x because the program will exit the program in the unlocked state if it enters in the unlocked state, no matter what x is. The summary now contains (c), (d), (f), and (i). Since we assume the initial state is unlocked, we eliminate (c), (d), and (f) and retain (i) as the final summary of the program segment.

5. RESULTS

We have implemented the translation described in Section 2, 3, and 4. Together, these comprise 682 lines of O'Caml code. In addition, in order to feed the resulting boolean formula into a conventional SAT solver, we convert the formula into conjunctive normal form (CNF), which essentially flattens the formula into a set of clauses. This step is fairly standard and follows the outline given in [6].

Table 1 shows some timing experiements we performed to see whether SAT is a viable back-end solver for formulas obtained from this translation. Our implementation has not been optimized. We use Chaff [18] as our back-end SAT solver. The tests were performed on a 1.1GHz dual Pentium III with 1GB of memory. The examples are all given in this paper, except for a version of Euclid's algorithm and a simple buffer overrun example from Dor's paper [11], described below.

5.1 Detecting Buffer Overruns

One serious class of computer security problems is caused by careless handling of string buffers in C. These bugs often allow a malicious attacker to take over control of a remote system, and this class of vulnerabilities is widely exploited.

Recently, researchers have attempted to use static analysis to attack this problem. Dor et. al. annotate string operations with integer variables capturing their effects on string buffers [11]. The annotated program contains assertions that are checked using an abstract interpretation algorithm that discovers linear relationships between variables.

We have manually annotated two examples from Dor's paper, converting the code into Simple. Our translation approach can detect the errors in the example programs with the simple assertion checker described in Section 3 and also with the slicing translator from Section 4[1]. Table 1 shows that we translated and solved the `insert_long` example in 2.84s, and SAT only took up 1.33s of that time. The time reported by Dor for this example is 38.8s (but our test machine is approximately three times as fast as the machine Dor used in her tests).

With the slicing translation, we were able to cut down the size of the final summary by a third from 186327 to 123206 clauses. We delayed the feasibility test in the Compose rule to the end of the translation. The total running time is 7.50 seconds, of which 4.49 seconds is spent in the SAT solver. It is slower than the simple translation because of repeated calls to the SAT solver in the process of merging the transitions, but the end result is a more concise summary of the program segment in terms of the FSM property we are checking.

6. RELATED WORK

Our work is based on the model used by Metal [12, 14], where properties are specified as state machines that pattern match specific source code to recognize when to perform transitions. We take the set of transitions for granted and perform an additional analysis step that determines if any of the paths that lead to an error are feasible paths. Previously, Metal used an algorithm based on congruence closure to perform false path pruning. That approach was unsound, could not handle as many types of conditions and expressions, and only worked on a single path at a time.

SLAM [3] and Lazy Abstraction [15] use an iterative technique that includes predicate abstraction, model checking, and counterexample-based refinement. One important advantage this technique enjoys is that it is interprocedural without the need for inlining. We differ from these approaches in that we do not use predicate abstraction to dynamically refine the set of relevant predicates. Instead, once we determine that an error may be possible, we directly transform all the program's statements and control flow into a single compact satisfiability problem. Our approach is guaranteed to terminate, but we lose precision with loops and recursive functions. On the other hand, our approach wastes little time recovering implications between predicates.

ESP [9] currently performs constant propagation to prune paths; more complex relationships between variables are not handled. In addition, because it is not demand-driven, it propagates information even if it is not needed. In contrast, we only convert a procedure to SAT if there is a potential path to an error. On the other hand, ESP is interprocedural and context sensitive, thanks to an interprocedural dataflow analysis framework [20].

This work bears the most similarity to the verification generation phase of Extended Static Checking [10] (ESC). ESC generates verification conditions for a procedure at a time. It requires pre/post condition annotations on functions to model side effects. Our Phase I, and assertion translations are similar to ESC's technique for generating compact verification conditions [13]. However, ESC does not use the idea of transitions to perform program slicing, as we do in Section 4.

Prefix [5] performs symbolic execution to capture fairly precise information about program behavior in order to detect common programming errors such as memory leaks and NULL pointer dereferences. They arbitrarily limit the number of paths they explore because of the expense of full symbolic evaluation.

Model checking [7] is an exhaustive state exploration technique that has become popular for hardware verification. A model must be created, which requires significant effort to produce an abstraction that is accurate enough to rule out false paths but small enough to be model-checked. Recently some work has been done on Bounded Model Checking (BMC), which is essentially model checking using SAT solvers. BMC has been shown to avoid the state explosion problem in some cases [6, 4]. In BMC, approach, the user provides an initial state $I(x_0)$ and a transition relation $T(x_i, x_{i+1})$, which are transformed into SAT formulas. SAT is called repeatedly with longer and longer formulas, each time adding a copy of the transition relation and therefore searching one level further from the initial state in the state space. The analogous technique for the approach in this paper would be to unroll loops iteratively.

Alloy [16] translates first-order relational logic into SAT to decide formulas. The translation handles quantifiers and

[1]We can model assertions by a simple FSM transition from state OK to Failure.

Example	LOC	Clauses	Vars	CNF	SAT	Total
Figure 3a	6	2938	615	0.03s	< 0.01s	0.03s
Figure 3b	3	1527	1011	0.01s	< 0.01s	0.01s
Euclid	8	5970	2545	0.05s	0.03s	0.08s
simple buffer overrun	143	42261	10641	0.34s	0.04s	0.38s
`insert_long` buffer overrun	508	186246	44470	1.51s	1.33s	2.84s

Table 1: Timing measurements on sample programs. LOC is lines of Simple code.

other constructs. Unlike this work, Alloy aims to check properties of specifications, although some work has been done on using Alloy to check program properties [17].

7. CONCLUSION

We have shown how to perform path-sensitive analysis by tranforming it into a boolean satisfiability problem. By integrating program slicing, finite state machine properties, and the ability to reason about values, we believe this analysis framework holds promise for error detection and verification in safety and security-critical code.

8. REFERENCES

[1] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[2] T.M. Austin, S.E. Breach, and G.S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, June 1994.

[3] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for the Analysis and Construction of Systems (number 1579 in LNCS)*, 1999.

[5] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[6] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. In *Formal Methods in System Design*, July 2001.

[7] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment from and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[9] Manuvir Das, Sorin Lerner, and Mark Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[10] D.L. Detlefs, R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. TR SRC-159, COMPAQ SRC, December 1998.

[11] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in c programs via integer analysis. In *Static Analysis Symposium (SAS)*, 2001.

[12] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.

[13] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th Annual Symposium on Principles of Programming Languages*, 2001.

[14] S. Hallem, B. Chelf, Y. Xie, and D.R. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[15] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, January 2002.

[16] D. Jackson. Automating first-order relational logic. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2000.

[17] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Internation Symposium on Software Testing and Analysis*, 2000.

[18] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *39th Design Automation Conference*, June 2001.

[19] G.C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[20] Thomas Reps, Susan Horowitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22th Annual Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[21] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages 3(3)*, pages 121–189, 1995.

[22] M.N. Velev and R.E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *Journal of Symbolic Computation, special issue on Integration of Automated Reasoning and Computer Algebra Systems*, 2002.

[23] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *The 2000 Network and Distributed Systems Security Conference. San Diego, CA*, February 2000.