Unit Checking: Symbolic Model Checking for a Unit of Code

Elsa Gunter[1] and Doron Peled[2]

[1] Department of Computer Science
New Jersey Institute of Technology
Newark, NJ 07102, USA
[2] Dept. of Computer Science
The University of Warwick
Coventry, CV4 7AL, UK

Abstract. We present a symbolic model checking approach that allows verifying a unit of code, e.g., a single procedure or a collection of procedures or methods that interact with each other. We allow temporal specification that asserts about both the program counters and the program variables. We decompose the verification into two parts: (1) a search that is based on the temporal behavior of the program counters, and (2) the formulation and refutation of a path condition, which inherits conditions on the program variables from the temporal specification. This verification approach is modular, as there is no requirement that all the involved procedures are provided. Furthermore, we do not require that the code is based on a finite domain. The presented approach can also be used for automating the generation of test cases for unit testing.

1 Introduction

Software errors are very hard to chase. The effort of tracing errors may sometimes surpass the effort of programming. The traditional bug-hunting technique is testing [6]. It is based on exercising the code in an attempt to manifest some errors. The testing process is usually performed by an experienced programmer, based on his/her experience. Testing is often targeted towards finding some common programming errors, such as division by zero or array reference out of range.

There are two main principles that guide testers in generating test cases. The first principle is *coverage*, where the tester attempts to exercise the code in a way that reveals maximal errors with minimal effort. The second principle is based on the tester intuition; the tester *inspects* the code in presuit of suspicious executions. In order to reaffirm or alleviate a suspicion, the tester attempts to derive the code through these executions.

In *unit testing*, only a small piece of the code, e.g., a single procedure or a collection of related procedures, is checked. It is useful to obtain some automated help in generating a test harness that will derive the appropriate executions. If we want to verify a software unit separately, we may code a *driver* that will

activate the checked code with some possible values, and *stubs*, which imitate the effect of missing procedures that are called by the tested unit. Generating a test condition can be done by calculating the path condition [2]. Coverage can be obtained by using various search algorithms through the flow control of the code. The main restriction here is that it is usually infeasible to obtain a comprehensive coverage.

Model checking [1] is a newer technique, which allows the automatic and systematic coverage of the code, provided that we have sufficient amount of memory and patience. It can be used to find some fixed properties such as deadlock in concurrent systems, or to systematically check for a given property. Model checking attempts to perform a comprehensive search, but it is limited by the size of the state space. A common restriction of model checking that we address in this paper is that it is usually applied to a fully initialized program, and assumes that all the procedures used are available.

In this paper, we describe a technique that allows the symbolic verification of a unit of code and the test case generator for it. The method we propose is based on model checking and theorem proving principles. The user gives a specification for paths along which a trouble seem to occur. The system automatically searches for possible executions that satisfy this specification, and symbolically calculates the path conditions. It suggests instantiations that can derive the execution through these paths.

A driver for the checked unit of code is replaced by providing an assertion on the relation between the variables at the start of executing the unit. Stubs for procedures not provided are replaced by further assertions, which relate the values of the variables at the beginning of the execution of the procedure with their values at the end. This allows us checking part of the code, rather than a complete system, hence achieves compositionality.

The advantages of our approach are:

- Combating state space explosion. Checking how some parts of the code behave with respect to many values, all at the same time. A reported path is represented in a parametric way in the sense that it is given with an initial condition and a sequence of program counters. This can correspond to multiple (even infinitely many) executions.
- Compositionality. Being able to check part of the code, rather than all of it.
- Infinite state space verification. In some cases, we may show correctness with respect to some unbounded parameters. Of course, some inherent undecidability affects the method, rendering it semiautomatic in ways that will be explained.
- The automatic generation of test cases, given as path conditions.

2 Interactive Unit Testing

Our proposed technique combines ideas from testing, verification and model checking. The architecture is shown in Figure 1. We first compile the program

into a flow chart. We keep separately the skeleton of the flow chart, abstracting away any variable. We also obtain a collection of *atomic transitions* that correspond to the basic nodes of the flow chart.

We specify the program paths that we suspect as having some problem (thus, the specification is given 'in the negative'). The specification corresponds to the tester's intuition about the location of an error. For example, a tester that observes the code may suspect that if the program progresses through a particular sequence of instructions, it may cause a division by zero. The tester can use a temporal specification to express paths. The specification can include assertions on both the program counters, and the program variables.

Our model checker generates paths that fit the restrictions on the program counters appearing in the specification and ignores for the moment the assertions on the program variables. Given a path, it uses the transitions generated from the code in order to generate the *path condition*. The assertions on the program variables that appear in the specification are integrated into the generated path condition, as will be explained below. The path condition describes values for the program variables that will allow or guarantee passing it through. Given a path, we can then instantiate the the path conditions with actual values so that they will form test cases. In this way, we can also generate test cases that consist of paths and their initial conditions.

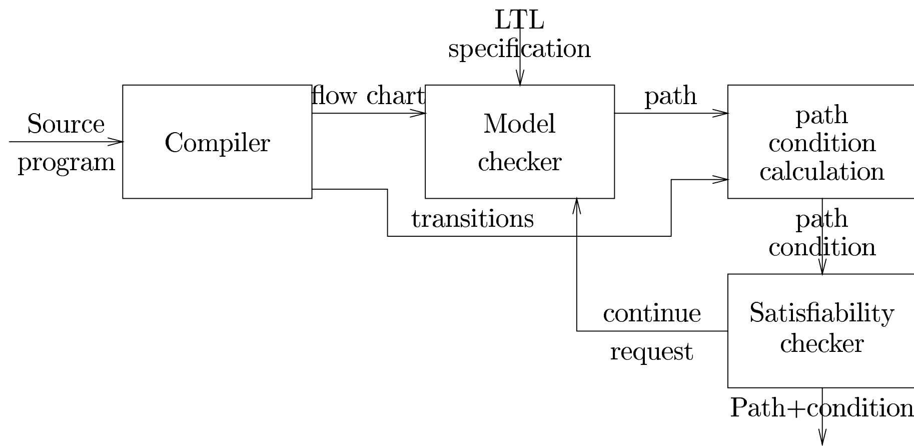

Fig. 1. Automatic test case generation

Passing control between the different units that participate in our design, we employ the model checker as a *coroutine*. We pass control to the model checker to continue the search from where it has stopped if the current path condition calculates to *false* or the user wants to obtain another path. For each prefix of a path generated, we calculate and simplify the path condition. If the path

condition is simplified to *false*, we reject it 'on-the-fly', i.e., we do not continue the search from this state, as this path cannot correspond to any legal execution. If it does not simplify to *false*, we can report it. (Note that simplifying a path condition to *false* in some domain is semi decidable. Therefore, we may report some paths that cannot be executed.)

Calculating the path conditions

There are two main possibilities in calculating the path conditions: *forward* and *backward*. The forward calculation is based on assigning symbolic values to the program variables, initially the same as the variable names. Moving forward in the path, we need to change the symbolic values when we perform an assignment of the form $x := e$. We look up the table of symbolic values, and replace each variable in e by its symbolic value. We then replace the symbolic value of x by the expression obtained. For example, consider the assignment $x := x * z$. If the current symbolic value of x is $y + 5$ and of z is $t + 2$, we obtain the expression $(y+5)*(t+2)$ (the parentheses are added in order to keep the order of calculation correct). This is the new value of x in the symbol table.

We also keep an accumulative path condition. This is the condition to pass from the beginning of the path to the current point in the path evaluation. The current point is an edge of the flow chart. It moves at each step in the calculation over one node to the subsequent edge. Initially this path condition is set to *true*. When we reach a condition, we replace the variables in the condition with their current symbolic expression and add that as a conjunct, or as a negated conjunct, depending on whether we exit the condition node with the *yes* or with the *no* edge, respectively. For example, if we reach a condition $x > z$, and we exit the condition with the *no* edge, we add to the path condition the conjunct $\neg(y + 5) > (t + 2)$. That is, for $\neg(x > z)$ to hold at this point, means that $\neg(y + 5) > (t + 2)$ holds at the beginning of the path. When the entire path is traversed, we report the accumulated path condition.

We can also calculate the path condition backwards. The accumulated path condition now represents the condition to move from the current point in the calculation to the end of the path. The current point moves at each point backwards over one node to the previous edge. We start again with a *true* condition. When we pass a condition edge (on our way back), we either conjoin it as is, or conjoin its negation, depending on whether we exit it with a *yes* or *no* edge, respectively. When we pass an assignment, we "relativize" the path condition φ with respect to it; if the assignment is of the form $x := e$, where x is a variable and e is an expression, we substitute e instead of each free occurrence of x in the path condition. This is denoted by $\varphi[e/x]$.

Consider the path in Figure 2. When we calculate the path condition in the forward direction, we start with variable x having the symbolic value x and y having the symbolic value y. The condition to pass the empty path so far is *true*. Arriving at the condition node, we replace the occurrence of x in $x > y$ with its current symbolic value $x + 1$, obtaining $x + 1 > y$. Since we took the exit edge labeled with *no*, i.e., the condition does not hold, we conjoin the negation

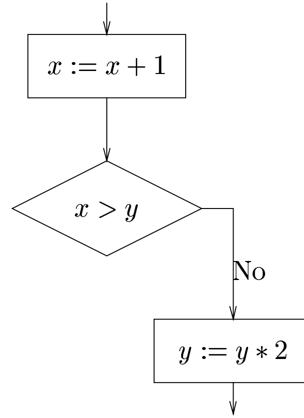

Fig. 2. A path

of this predicate, obtaining (after simplification, removing the redundant *true* conjunct), $\neg x+1 > y$. The next assignment causes replacing the symbolic value of y with $y * 2$, but the path condition remains the same.

Calculating the path condition for this example backwards, we start at the end of the path with a path condition *true*. Moving backwards through the assignment $y := y * 2$, we may want to substitute every occurrence of y with $y * 2$. However, there are no such occurrences in *true*, so we remain with *true*. Progressing backwards, we now conjoin the negation of the condition, which is $x > y$, obtaining $\neg x > y$. This is now the condition to execute the rest of the path. But we still have to relativize it to the assignment $x := x+1$, which means replacing the x with $x + 1$, obtaining again $\neg x + 1 > y$.

The choice of direction for calculating the path condition is affected by the direction in which the path is obtained. We may also want to calculate the path conditions incrementally for the prefixes of that path, or the suffixes. The reason is that path condition calculation is rather expensive, and thus we may benefit from discarding a prefix (or a suffix, respectively) 'on-the-fly', recognizing that its path condition is *false*. We then save the effort of calculating paths with that prefix (or suffix, respectively).

Translating the Specification

We limit the search by imposing a property of the execution path we are interesting in. The property may mention the labels that our path passes through and some relationship between the program variables. It can be given in various forms, e.g., regular expressions, an automaton or a temporal formula. We are only interested in properties of finite sequences; checking for cycles is, in general, impossible, since we cannot identify repeated states.

The specification includes the following two types of *basic* formulas:

Program counter predicates, of the form *at* l, where l is a program counter label. In translating the checked code into a flow chart for a visual representation, we automatically generate a label for each node, and can use that node in the specification.

Program variables assertion. A predicate that includes the program variables (and does not include further Boolean operators).

These formulas may be combined using Boolean and temporal operators. The Linear Temporal Logic (LTL) syntax is as follows:

$$\varphi ::= (\varphi) \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \overline{\bigcirc}\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \varphi\, \mathcal{V}\, \varphi \mid p$$

where $p \in \mathcal{P}$, with $\mathcal{P}$ a set of basic formulas. We denote a propositional sequence over $2^{\mathcal{P}}$ by σ, its ith state (where the first state is numbered 0) by $\sigma(i)$, and its suffix starting from the ith state by $\sigma^{(i)}$. Let $|\sigma|$ be the length of the sequence Σ, which is a natural number. The semantic interpretation of LTL is as follows:

- $\sigma \models \bigcirc\varphi$ iff $|\sigma| > 1$ and $\sigma^{(1)} \models \varphi$.
- $\sigma \models \varphi\, \mathcal{U}\, \psi$ iff $\sigma^{(j)} \models \psi$ for some $0 \leq j < |\sigma|$ so that for each $0 \leq i < j$, $\sigma^{(i)} \models \varphi$.
- $\sigma \models \neg\varphi$ iff it is not the case that $\sigma \models \varphi$.
- $\sigma \models \varphi \vee \psi$ iff either $\sigma \models \varphi$ or $\sigma \models \psi$.
- $\sigma \models p$ iff $|\sigma| > 0$ and $\sigma(0) \models p$.

The rest of the operators can be defined using the above operators. In particular, $\overline{\bigcirc}\varphi = \neg\bigcirc\neg\varphi$, $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$, $\varphi\, \mathcal{V}\, \psi = \neg((\neg\varphi)\, \mathcal{U}\, (\neg\psi))$, $\mathit{true} = p \vee \neg p$, $\mathit{false} = p \wedge \neg p$, $\Box\varphi = \mathit{false}\, \mathcal{V}\, \varphi$, and $\Diamond\varphi = \mathit{true}\, \mathcal{U}\, \varphi$. The operator $\overline{\bigcirc}$ is a 'weak' version of the $\bigcirc$ operator. Whereas $\bigcirc\varphi$ means that φ holds in the suffix of the sequence starting from the next state, $\overline{\bigcirc}\varphi$ means that *if* the current state is not the last one in the sequence, *then* the suffix starting from the next state satisfies φ.

We distinguish between the operator $\bigcirc$, which we call *strong nexttime*, and $\overline{\bigcirc}$, which we call *weak nexttime*. Notice that

$$(\bigcirc\varphi) \wedge (\overline{\bigcirc}\psi) = \bigcirc(\varphi \wedge \psi)$$

since $\bigcirc\varphi$ already requires that there will be a next state. Another interesting observation is that the formula $\overline{\bigcirc}\mathit{false}$ holds in a state that is in deadlock or termination.

The operators $\mathcal{U}$ and $\mathcal{V}$ can be characterized using a recursive equation, which is useful for understanding the transformation algorithm, presented in the next section. Accordingly, $\varphi\, \mathcal{U}\, \psi = \psi \vee (\varphi \wedge \bigcirc\varphi\, \mathcal{U}\, \psi)$ and $\varphi\, \mathcal{V}\, \psi = \psi \wedge (\varphi \vee \overline{\bigcirc}(\varphi\, \mathcal{V}\, \psi))$.

The specification is translated into a finite state automaton. The algorithm is the one described in [3], and relativised to finite sequences, as in [2], with further optimizations to reduce the number of states generated. Let (S, Δ, I, F, L) be a finite state automaton with states S, a transition relation $\Delta \subseteq S \times S$, initial states $I \subseteq S$, accepting states $F \subseteq S$ and a labeling function L from S to some

set of labels. The *property automaton* is $A = (S^A, \Delta^A, I^A, F^A, L^A)$. Each property automaton node is labeled by a set of basic formulas. The flow chart can also be denoted as an automaton $B = (S^B, \Delta^B, I^B, S^B, L^B)$ (where all the states are accepting, hence $F^B = S^B$). Each node in S^B is labeled by a single program counter. The intersection is an automaton $A{\times}B = (S^{A\times B}, \Delta^{A\times B}, I^{A\times B}, S^{A\times B}, L^{A\times B})$. The states $S^{A\times B} \subseteq S^A \times S^B$ have matching labels: the program counter of the flow chart must satisfy the program counters predicates labeling the property automaton nodes. The transitions are $\{((a,b),(a',b'))|(a,a') \in \Delta^A \wedge (b,b') \in \Delta^B\} \cap (S^{A\times B} \times S^{A\times B})$, We also have $I^{A\times B} = (I^A \times I^B) \cap S^{A\times B}$, and $F^{A\times B} = (F^A \times S^B) \cap S^{A\times B}$.

One intuition behind the use of a formula to constrain the search is that a human tester that inspects the code usually has some suspicion about some execution paths. For example, a path that passes through label l_2 exactly twice seem to lead to some incorrect use of resources. We may express this in LTL as

$$\neg l_2 \mathcal{U}(l_2 \wedge \bigcirc(\neg l_2 \mathcal{U} l_2)). \tag{1}$$

This formula can be translated to the property automaton that appears on the left in Figure 3.

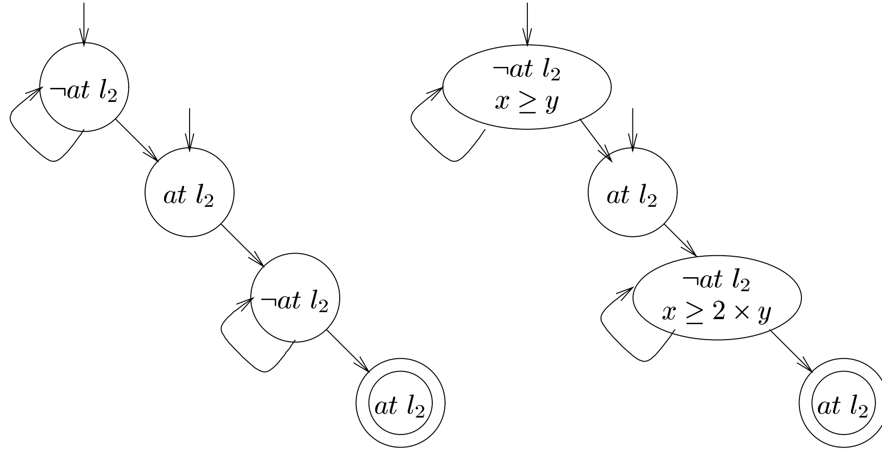

Fig. 3. A property automaton

The example in Figure 4 shows why must avoid identifying flow chart node with the same program counter that appear in a path as being the same. Suppose that after being in label l_2, the search gives priority to label l_3, rather than to l_1. A DFS without the capability of comparing states will search only through the loop containing l_2 and l_3. On the other hand, if we assume that the states with the same label are equivalent, we would not allow, e.g., the path $l_2 l_3 l_2 l_3 l_2 l_1$ (since it contains the nodes l_2 and l_3 more than once), which might be the one

with the error. Thus we may not assume that two nodes in the flow chart with the same program counter are the same, as they may differ because of the value of the program values. We may also not assume that they are different, since the values of the program variables, which are not included in the flow chart, may after all be the same.

Our solution is to allow the user to specify a limit n on the number of repetitions that we allow each flow chart node, i.e., a node from S^B, to occur in a path. We keep and update for each state found an additional field that counts the number of times this state have appeared on the DFS search stack so far. If this value is smaller than n, we allow yet another copy. Since our specification is based on finite sequences, we do not loose information by failing to identify cycles. Repeating the model checking while incrementing n, we eventually cover any length of sequence. Hence, in the limit, we cover every path. But this is of course impractical.

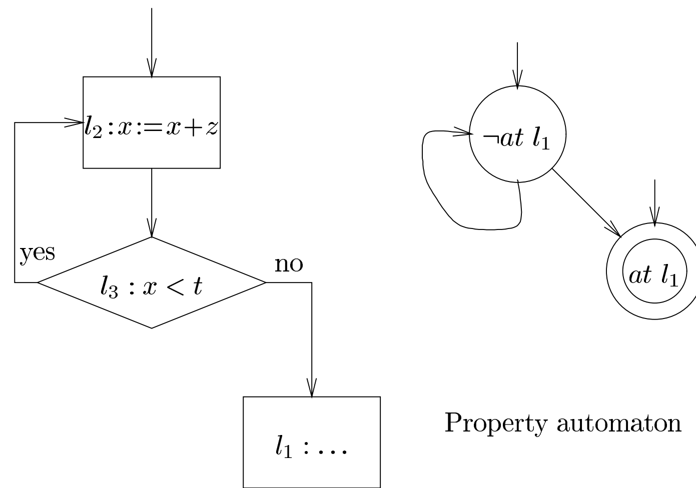

Fig. 4. A problem with identifying repeated nodes

By not identifying when states are the same, we may run into a combinatorial explosion. For example, consider the state space in Figure 5. Because we cannot identify when we reach the same node, we treat states that are reached from different directions as different. Then, at worst, the number of paths can be explored is exponential with the number of diamonds. Again, in order to reduce the amount of work done, we calculate the path conditions incrementally, backtracking from a prefix when it is calculated to *false*.

The specification formula (1) is based on on the program counters. Suppose that we also want to express that when we are at the label l_2 at the first time, the value of x is greater or equals that of y, and that when we are at the label

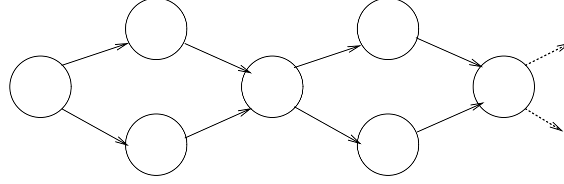

Fig. 5. An example with exponentially many sequences

l_2 the second time, x is at least twice as big as y. We can write the specification as follows:

$$\neg l_2 \mathcal{U}((l_2 \wedge x \geq y) \wedge \bigcirc(\neg l_2 \mathcal{U}(l_2 \wedge x \geq 2 \times y))). \quad (2)$$

The program variables assertions $x \geq y$ and $x \geq 2 \times y$ are added to the second and fourth nodes of the automaton. The automaton obtained appears on the right in Figure 3. They do not participate in the automata intersection, hence do not contribute further to limiting the paths. Instead, they are added to the path condition in the appropriate places.

If we apply the property automaton in Figure 3 to the flow graph on the left in Figure 4, we may obtain a path $l_2 l_3 l_2$. Calculating the path condition for the original property (1) backwards starts with setting $\varphi =$*true*. Because of the assignment in l_2 (which is the last node in the path), we relativize φ, but still remain with *true*. We now add as conjunct the condition in l_3, obtaining $x < t$. Because of the node l_2 (this time its first occurrence in the path), we relativize φ and obtain $x + z < t$.

Consider now the property (2). According to the matching, the condition $x \geq 2 \times y$ should be added just before the second occurrence of l_2, and $x \geq y$ should occur before the first occurrence. Again, calculating the path condition backwards, we start with *true*. Relativizing φ to the assignment in l_2 maintain *true*. At this point, we add the conjunct $x \geq 2 \times y$, obtaining $\varphi = x \geq 2 \times y$. We reach the node labeled with l_3, and add the conjunct $x < t$, obtaining $x \geq 2 \times y \wedge x < t$. We now reach node l_2, relativizing again with respect to the assignment $x := x + z$, we obtain $x + z \geq 2 \times y \wedge x + z < t$. At this point we also need to add the conjunct $x \geq y$ from the property automaton, obtaining the condition $x + z \geq 2 \times y \wedge x + z < t \wedge x \geq y$.

In general, the conjunction of the program variables assertions appearing in the property automaton node matching the current flow chart node is be added to the path condition *before* the effect of the path node which is associated with it. Therefore, if we add a condition η from some property automaton node to an assignment node in the flow graph, the assignment will take effect right after η has to hold.

It is important to observe that the LTL to automata translation generates nodes that are labeled by a set of basic formulas, either negated or non negated. A flow chart that matches such a node must satisfy the conjunction of these formulas. Our separation of the search depends on the fact that we do not allow

any basic formula that includes both program variables and program counters, as in $(at\ l_3) \times v$. Such formulas are used, e.g., in [7], and can usually be translated (unfortunately with some increase to the size of the formula) into formulas that make the required separation.

Incorporating Specifications for Missing Code

As with unit testing, when we want to check a unit of code, we may need to provide drivers for calling the checked procedure, and stubs simulating the procedures used by our checked code. Since our approach is logic based, we can use a *specification* for drivers and stubs, instead of using their code.

We replace the stubs with assertions about the relation of the program variables before and after the execution of the stub procedure. The variables before the execution of the procedure retain their original names, while the versions after the execution are primed, e.g., x' represents the value of x after the execution of a procedure. We also allow the predicate *same*, with a set of variables as an argument, to denote that the mentioned variables are not changed. For example, our procedure call may be expressed using a single flow chart node that asserts that $same(x, y) \wedge z' = z + 1$. This means that the variables x and y retain their value between the procedure call and its termination, while z is incremented. Variables that do not appear may obtain any value during the execution of the procedure. The above use of the predicate *same* is a syntactic sugar for $x' = x \wedge y' = y$.

We can easily incorporate such a procedure call into our calculation of the path condition. We start with the backwards path calculation. Let $\varphi(V)$ be the property for the rest of the path after the procedure call, where V is a list of program variables, and $\eta(V, V')$ is the formula expressing the effect of the procedure, as exemplified above, where V' is the variables V' primed. The path condition becomes $\exists V'(\varphi[V'/V] \wedge \eta)$. (If $V = \{x, y\}$, this notation is a shorthand for $\exists x \exists y(\varphi[x'/x, y'/y])$, where multiple substitution is done simultaneously.) That is, the formula obtained by conjoining η to a version of φ where each variable is replaced by its primed version; Then the primed variables are eliminated by existential quantification. Although we introduce existential quantification, many of the these quantifications are easily eliminated.

The forward calculation of the path condition is more complicated when stubs are used. We cannot keep the symbolic values of each variable since their change is given as a logical formula instead of an assignment. Instead, we will keep a predicate $\varphi(V, V')$, which connects the values of the variables V before the beginning of the path with the variables V' at the current point of the path. The variables V' are the primed version of the variables V. As we progress forward in calculating the path condition, we update this predicate. When we traversed the entire path, the returned path condition is $\exists V'\, \varphi(V, V')$.

The following considerations are needed to handle the annotations of the property automaton. Suppose we have calculated the path condition $\varphi(V)$ is the accumulated path, and now pass backwards over a matching pair of nodes from the property automaton and the flow chart. We *first* handle the flow chart

node, as explained above. To handle the property automaton node, labeled with a set of basic formulas, let the conjunction of the program variables assertions be $\mu(V)$. Then the path condition becomes now $\mu \wedge \varphi$.

We start with $\varphi(V, V') = (V = V')$. There are three kinds of nodes:

Condition node, with condition $\eta(V)$. We transform $\varphi(V, V')$ into $\varphi \wedge \eta[V'/V]$.

A stub node with a condition $\eta(V, V')$. The new path condition is transformed into $\exists U(\varphi[U/V'] \wedge \eta[U/V])$, with new variables U.

Assignment $x := e$. This can be treated as a stub with an assertion $x' = e$, and a *same* predicate that includes all the program variables, except for x. Correspondingly, the resulted condition, after eliminating some existential quantifiers, is $\exists t\,(\varphi[t/x'] \wedge x' = e[t/x])$, with a new variable t.

In addition, we have to handle the annotations of the property automaton. Again, suppose we have calculated the path condition $\varphi(V, V')$ of a prefix of the path, and now pass over a matching pair of nodes from the property automaton and the flow chart. Let the property automaton node be labeled with a set of basic formulas, where the conjunction of the program variables assertions is $\mu(V)$. Then the path condition becomes now $\mu[V'/V] \wedge \varphi$. That is, the variables in the condition μ are renamed to refer to the current set of variables V', and the obtained condition is added to the accumulated path condition. Note that we perform this transformation of the path condition *before* we perform a transformation related to the new flow chart node.

The forward calculation, in the presence of stubs, becomes considerably more complicated, as it involves introducing new existential quantifiers, which we aim to eliminate whenever possible. As mentioned, the forward calculation is useful to eliminate some paths 'on-the-fly', i.e., when the prefix of the path is reduced to *false*. A repeated backwards calculation of the prefixes of the paths can induce a quadratic increase in complexity (the sum of the lengths of all prefixes of a path is proportional to the length of the path squared). However, it is not clear that in this case the forward calculation of the path condition is more efficient than repeating the backwards one for each prefix.

We replace a driver by an initial condition that expresses the relation between the program variables at the beginning of the execution as a first order formula Θ. Accordingly, if the checked temporal condition is μ, we check $\Theta \wedge \mu$.

3 Example

Consider the Euclid greatest common divisor (gcd) algorithm in Figure 6. The initial condition for this procedure $\Theta = a > 0 \wedge b > 0 \wedge at\ l_0$. The expression $x\ rem\ y$ denotes the reminder of dividing x by y. At the end, the value of x should be the gcd of a and b. The algorithm uses a procedure to calculate the reminder of x divided by y into the variable z. The labels are generated automatically by compiling the program into a flow chart, as done by the PET system [2]. An error is introduced into this presentation, as the nodes labeled by l_4 and l_5 are reversed from the correct algorithm.

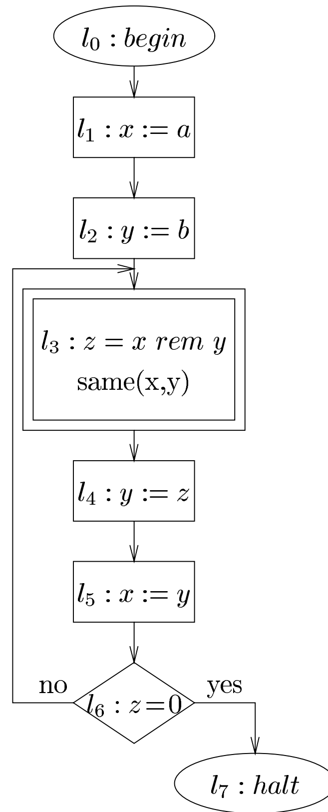

Fig. 6. Euclid's gcd algorithm with an error

We suspect that we may end up with a wrong result: instead of the gcd, the value of x at the end is 0. We use the specification $\diamond(at\ l_7 \wedge x = 0)$ to check this suspicion. This is conjoined with the initial condition to form $a > 0 \wedge b > 0 \wedge at\ l_0 \wedge \diamond(at\ l_7 \wedge x = 0)$. The property automaton appears in Figure 7.

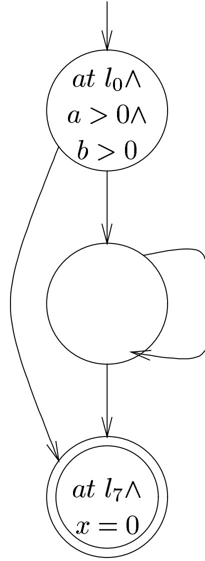

Fig. 7. A property automaton with two kinds of labels

Now, suppose that we perform verification, setting n, the number of times we allow each state to iterate, to 1. The intersection of the property automaton and the flow chart graph, projected on the flow chart, is a path $l_0 l_1 l_2 l_3 l_4 l_5 l_6 l_7$. The intersection does not take yet into account the basic program variable properties $a > 0$, $b > 0$ and $x = 0$, and is based on matching the labels only.

Now, we form the path condition for the above path. But we also add for each node in the path the condition from the matching node in the property automaton. This forces the path to satisfy at the beginning the condition $a > 0 \wedge b > 0$ and at the end the condition $x = 0$. We obtain a condition that can be simplified to $a\ rem\ b = 0$, i.e., within one iteration, we terminate with $x = 0$ if b divides a (we are dealing with integer arithmetic). If we repeat this with $n = 2$, we obtain also another sequence, $l_0 l_1 l_2 l_3 l_4 l_5 l_6 l_3 l_4 l_5 l_6 l_7$. The path condition, when simplified, is equivalent to $a\ rem\ b \neq 0$. This means that when b does not divide a. we terminate with $x = 0$ after two traversals of the loop.

We can stop now, since in any case, either $a\ rem\ b =$ or $a\ rem\ b \neq 0$. Thus, we terminate with $x = 0$, either after one or two executions of the the loop. We obtained two generic paths that cover *all* the cases, corresponding to infinitely many executions according to the different values of a and b. Note that if we did

not observed that the disjunction of these path conditions covers all the cases, we may continue with $n = 3$, $n = 4$, etc. We will not find any more paths. There are paths that match the property automaton, restricted to the program counter label. But these paths have a path condition *false*. Note however that in general, we are not going to be warned that there are no more paths and that we have covered all the cases.

4 Discussion

We proposed and implemented a symbolic verification approach for a unit of code, which we call *unit checking*. Our approach allows the verification of temporal properties, which reference both the program counters and the program variables. The verification search includes two components: abstracting away the program variables, while performing the search on the flow chart, and calculating and refuting (i.e., attempting to simplify to *false*) path conditions, where assertions on the program variables are added to the path condition. These two parts work in accordance, as coroutines. That is, a model checker is performing the search on the flow chart, and then a path condition is calculated (incrementally). The control returns to the model checker after the calculation of the path condition when either the path is not complete, or the path condition was refuted.

Our model checking approach is semiautomatic in two ways:

1. Refuting the path condition is, in general, undecidable [5]. It is decidable for certain specific domain, e.g., for finite domains and for Presburger Arithmetic. We apply procedures for simplifying formulas, and for refuting Presburger arithmetic [8]. Still, because of the undecidability, the system may report a path condition that is equivalent to *false*.
2. As pointed out, we cannot compare states during the symbolic evaluation. We apply a strategy of putting a constraint on the length of the admitted paths. We apply DFS (or, alternatively, BFS) with this constraint, provided by the user. In the limit, we can cover any length of path, but there is no generic decision procedure that provides, for a given verification problem, consisting of a unit of code and a temporal specification, a limit to the size of the path.

Our approach can be used for the temporal verification of sequential pieces of code. It can also be extended to the verification of concurrent code. This requires adding programming constructs for concurrency, and adapting the path condition calculation to these constructs. Orthogonal work, which generlizes the path condition to analyze also pointer analysis, concurrency and aliasing, appears in [4]. That work, also focused on the symbolic evaluation of code, applies a model checker to derive different pointer aliasing. It is an interesting and nontrivial challenge to combine these two techniques.

Our approach can also be used for automating the unit testing process. Accordingly, a tester uses a formula to focus on some suspicious paths. The algorithm given in this paper is used to generate these paths and calculate the

corresponding path conditions. These can be used to exercise the code in order to confirm or refute the suspicion.

References

1. E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2000.
2. E.L. Gunter, D. Peled, Temporal Debugging for Concurrent Systems, TACAS 2002, Grenoble, France, LNCS 2280, Springer, 431-444.
3. R. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, *PSTV95, Protocol Specification Testing and Verification*, 3–18, Chapman & Hall, 1995,
4. S. Khurshid, C. Pasareanu, D. Peled, W. Visser, Generalized Symbolic Execution for Testing and Model Checking, in preparation.
5. Y. Matiyasevich, Hilbert's Tenth Problem, MIT Press, 1993.
6. G.J. Myers, The Art of Software Testing, John Wiley and Sons, 1979.
7. Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1991.
8. D. C. Oppen, A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. JCSS 16(3): 323–332 (1978).