DOI 10.1007/s00165-005-0059-8
BCS © 2005
Formal Aspects of Computing (2005) 17: 201–221

Formal Aspects of Computing

Model checking, testing and verification working together*

Elsa Gunter[1, 3] and Doron Peled[2, 4]

[1]Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, IL 61801-2302, 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 that interact with each other. We allow temporal specifications that assert over 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 constraining the *program variables* from the temporal specification. This verification approach is modular, as we do not require that all the involved procedures are provided. Furthermore, we do not request 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.

Keywords: Model checking; Test generation; Verification

1. Introduction

Software errors are very hard to trace. The effort of tracing errors may sometimes surpass the effort of programming. The traditional bug-hunting technique is testing [Mye79]. It is based on exercising the code in an attempt to manifest some errors. The testing process is usually performed by a veteran programmer, based on his/her experience. Testing is often targeted towards finding some common programming errors, such as division by zero or array reference being out of range.

There are two main principles that guide testers in generating test cases. The first principle is *coverage* [RP85], where the tester attempts to exercise the code in a way that reveals maximal errors with minimal effort. The second principle is to use the tester's *intuition* in order to *inspect* the code in pursuit of suspicious executions. In order to reaffirm or alleviate a suspicion, the tester attempts to exercise 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 tested. It is useful to obtain some automated help in generating a *test harness* that will exercise the appropriate executions. If we want to test 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

Correspondence and offprint requests to: Doron Peled, Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK. Email: doron@dcs.warwick.ac.uk
*A preliminary version of the paper, with the title *Unit Checking: Symbolic Model Checking for a Unit of Code* appears in the Lecture Notes in Computer Science volume 2772, *Verification – Theory and Practice*, celebrating Zohar Manna's 64th birthday.
[3]This research was partially supported by US Army Research Office Grant number DAAAD19-01-1-0473
[4]This research was partially supported by Subcontract UTA03-031 to The University of Warwick under University of Texas at Austin's prime National Science Foundation Grant #CCR-0205483.

tested unit. Generating a test condition can be done by calculating the path condition [GP02]. Coverage can be obtained by using various search algorithms through the flow chart of the code. It is usually infeasible to obtain a completely comprehensive coverage.

Model checking [CGP00] is a newer technique than testing, allowing the automatic and systematic coverage of the code. 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 it can handle. Common restrictions of model checking that we address in this paper are that (1) it is usually applied to a fully initialized program, (2) it assumes that all the procedures used are available, and (3) it usually handles systems with finite state spaces.

In this paper, we describe a technique we call *unit checking* that allows the symbolic verification of a unit of code and the generation of test cases. The method we propose is based on a combination of model checking and theorem proving principles. The user gives a specification for paths along which a trouble seems to occur. Unit checking automatically searches the paths of the flow chart of a program for possible executions that satisfy the specification. It symbolically calculates the path conditions and suggests instantiations that can derive the execution through these paths.

We allow a temporal specification based on both program counters and program variables. Our method separates the specification in such a way that a finite component of both the checked code and the temporal specification is intersected, as in state-based model checking. We apply on-the-fly (i.e., while performing the model checking) a symbolic calculation of the conditions to execute the paths in the intersection. We also apply automated simplification of the path condition, using various heuristics and decision procedures from a theorem prover. If the condition to execute the current path is simplified to *false*, the current path cannot be executed, and we refute it, backtracking our search.

A unit of code needs to work in the presence of other parts of code: the program that calls it, and the procedures that are called from it. In order to check a unit of code, we need to provide some representation for these other parts. Testing prescribes replacing the calling procedure by some simplified abstract code, called a *driver* and similarly, the called procedures by *stubs*. A driver is replaced in our approach by providing an assertion on the relation between the variables at the start of executing the unit. Stubs 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 to check parts of the code, rather than a complete system at once. The advantages of our approach are:

- *Combating state space explosion.* Checking how some parts of the code behave with respect to multiple values simultaneously. 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.
- *Modularity.* Being able to check part of the code in isolation, rather than all of it at once.
- *Parametric and infinite state space verification.* In some cases, we can show correctness with respect to some unbounded parameters, e.g., unbounded initial values to the program variables. Of course, some inherent undecidability affects the method, rendering it semiautomatic in ways that will be explained.
- The automatic generation of test cases from generated path conditions.

Related work includes [BGP99], which suggests model checking infinite state spaces by using symbolic executions. Since the specification is given there using the logic CTL, the model checking is done there through a fixpoint calculation. That paper also suggests heuristics for attacking the inherent undecidability of the model checking problem. The use of LTL over infinite state spaces is studied in [MP91a, MP91b], where a manual proof rule in the Manna-Pnueli style is suggested. The work of [KPV01] proposes a method that uses finitary abstraction to convert the infinite state space problem (when possible) into a finite instance of model checking. A recent paper [GLSU02] suggests to use temporal specification to obtain a desired coverage of extended finite state machines for testing. However, this paper does not deal with the generation of path conditions, and the temporal formulas used are based on static information in the nodes, rather than on program variables values.

The closest work to ours that we are aware of is the ESC/Java tool [FLLN02], where the code of a Java program is annotated with correctness assertions. The tool is used to automatically check these assertions. This is done by propagating the conditions backwards. Like in our approach, ESC/Java also allows replacing a procedure by its correctness assertion, in order to obtain modularity in the verification, in the style of [GL80]. However, that tool deals with static and fixed properties, rather than with verification of temporal specification. Our approach looks at the selection of paths driven by a temporal formula, rather that trying to verify some safety properties of the code.

2. Interactive unit checking

A *state* of a program is a function assigning values to every program variable according to their defined domains (e.g., integers, strings, fixpoint). *Augmented states* also include assignments of values to the program counters. We denote the fact that a state g satisfies a first order assertion φ by the standard notation $g \models \varphi$.

A flow chart of a program or a procedure is a graph, with nodes corresponding to assignments and tests taken, and edges reflecting the flow of control between the nodes. A flow chart can be obtained by automatic compilation of the code. There are several kinds of nodes, where the most common are a *box* containing an *assignment*, a *diamond* containing a *condition*, and an *oval* denoting the *beginning* or *end* of the program (procedure). Edges exiting from a diamond node are marked with either 'yes' or 'no' to denote the success or failure of the condition, respectively. We assume that each node has a unique program counter value. This value can be a label that is provided with the code, or automatically generated by a translation tool. Passing an edge out of one node and into another entails a corresponding change of the program counter value. A *path* of a program is a consecutive sequence of nodes in the flow chart.

An *execution* is a sequence of states $g_1\, g_2\, \ldots\, g_n$, where each state g_{i+1} is obtained from its predecessor g_i by executing a transition associated with a flow chart node, as described below. This means that the condition for the transition to execute holds in g_i, and the transformation associated with the transition is applied to it. We call an execution that contains augmented states an *augmented execution*. The projection of an augmented execution on the program counter values results in program counter values (labels) along a path in the flow chart. Thus, in general, a path may correspond to multiple executions, with the same sequence of program counter values.

Let $\tau = t_1, t_2, \ldots t_n$ be a path in a flow chart. Let $\rho = g_1, g_2, \ldots, g_n$ be a sequence of non augmented program states (i.e., not including the program counter values). The Sequence ρ is an *execution* of τ if for each $1 \leqslant i < n$ we have:

1. t_i is a diamond node, with condition c and the edge from t_i to t_{i+1} is marked with "yes". Then we must have $g_i \models c$ and $g_{i+1} = g_i$.
2. t_i is a diamond node, with condition c and the edge from t_i to t_{i+1} is marked with "no". Then we must have $g_i \models \neg c$ and $g_{i+1} = g_i$.
3. t_i is an assignment node labeled $x := e$. Then $g_{i+1} = g_i[x/e[g_i]]$, which denotes that g_{i+1} is the same as g_i, except for the variable x, which has the value of e, interpreted according to the state g_i.

We denote the executions of τ by $exec(\tau)$. Note that t_n is used only for the choice of the edge out of t_{n-1}. This is only important if the latter node is a condition, hence the choice of outgoing edge affects whether the condition holds or does not hold.

A *path condition* is a first order predicate that expresses the condition to execute the path, when starting from a given node (a path condition does not refer to the program counter values). Denote the path condition for a path τ by Γ_τ. Formally, this means that when starting executing from the first node t_1 of τ, every execution ρ of τ must begin with a state g_1 (not necessarily a unique state for all the executions) such that $g_1 \models \Gamma_\tau$. Moreover, when starting the execution at node t_1 with a state satisfying Γ_τ, we can extend g_1 to an execution ρ of τ. In deterministic code, there is at most one way of extending a state into an execution sequence. Hence, in this case, *all* the executions that start with a state satisfying Γ_τ are executions of the path τ or paths with prefix τ. In this paper we mainly concentrate on sequential and deterministic code, although our implementation also supports concurrency. In concurrent or nondeterministic code this is not guaranteed; this is further discussed in Sect. 6.

2.1. Architecture

Our proposed technique combines ideas from testing [Mye79], verification [Dij75] and model checking [CGP00]. The architecture is shown in Fig. 1. We first compile the program into a flow chart. We keep separately the structure of the flow chart, abstracting away all the variables. We also collect information about the contents of the nodes of the flow chart, to be used in calculating path conditions.

We specify the program paths that are suspected of 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 counter values (program location labels), and the program variables.

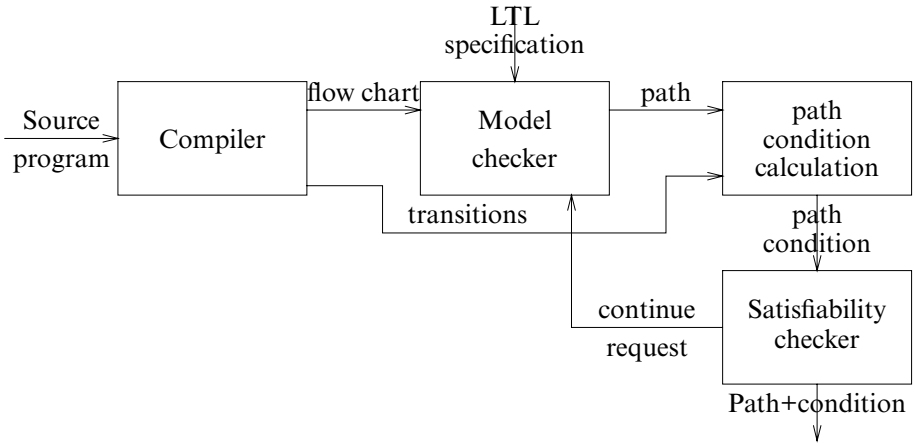

Fig. 1. Automatic test case generation

Our model checker generates paths that fit the temporal restrictions on the program counters appearing in the specification. Given a path, it uses the transitions obtained by translation 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 prescribes values for the program variables that will *guarantee* (in the sequential case, or *allow*, in the nondeterministic case, e.g., due to concurrency) passing through the path. We can then instantiate a path condition with actual values so that they will form a test case.

Passing the 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 in the following cases: (1) the current path may be extended, (2) the path condition can be simplified to *false*, and thus backtracking is required, or (3) the user wants to obtain an alternative path (and test case). 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 in some domains, e.g., the integers with multiplication, simplifying a path condition to *false* is semidecidable [Mat93]. Therefore, we may report some paths that cannot be executed.

Let τ be a path of the flow chart, σ an execution of τ, and φ_τ the path condition. Then the following can be proved by simple induction on the length of the path/execution (see e.g., [Pel02]): given that control is just before the first node, then the path τ will be executed. If nondeterminism is allowed, then τ will be executed provided that nondeterministic choices are resolved in a way that follows that path.

2.2. Calculating the path conditions

There are two main possibilities in calculating path conditions: *forward* [King96] and *backward* [Dij75]. We will explain both and describe their advantages and disadvantages.

Forward calculation

The forward calculation of the path condition is based on assigning symbolic values to the program variables, initially the same as the variable names. A *point* is an edge of the flow chart. The symbolic value of a variable at a certain point in a path reflects its current value expressed with respect to the value of variables at the beginning of the path. Thus, if the current symbolic value of a variable x is $y + z$, it means that currently the value of x is the sum of the values that y and z had at the beginning of the path. Progressing 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

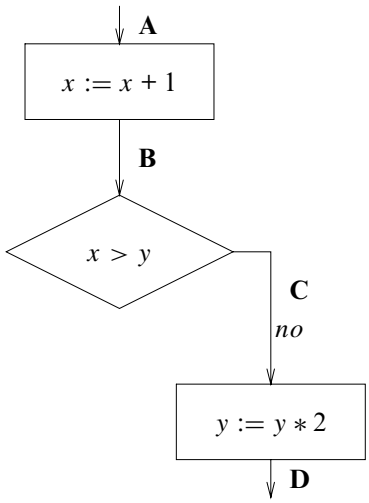

Fig. 2. A path

$y + 5$ and of z is $t + 2$, we obtain from $x * z$ the expression $(y + 5) * (t + 2)$ (the parentheses are added in order to keep the order of calculation correct). This becomes the new value of x in the symbol table.

We keep an *accumulated path condition*. This is the condition to pass from the beginning of the path to the current point in the path. Like the symbolic values, the accumulated path condition is also expressed with respect to the values of the variables at the beginning of the path. It can change when we pass from one node to the subsequent one. Initially this path condition is set to *true*. When we pass a diamond node, 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, under the above symbolic assignment, where x is $y + 5$ and z is $t + 2$, if we pass a condition node labeled with 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.

Consider the path in Fig. 2, which starts at point (edge) **A** and ends at point **D**. When we calculate the path condition in the forward direction, we start at point **A** 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*. We progress now to point **B** in the path. Passing the assignment $x := x + 1$ results in the symbolic value of x being replaced with the value $x + 1$. Passing the condition node to point **C**, 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 of this predicate, obtaining (after simplification, removing the redundant *true* conjunct) $\neg(x + 1 > y)$. This is the accumulated path condition to execute the path from point **A** to point **C**. The next assignment we pass, from **C** to **D**, causes replacing the current symbolic value of y with $y * 2$. The accumulated path condition remains the same.

Backwards calculation

We can also calculate the path condition backwards. We do not need to keep the symbolic values of the variables when propagating backwards. The accumulated path condition in this case represents the condition to move from the current point in the calculation to the *end of the path*. The current point moves at each step in the calculation of the path condition backwards, over one node to the previous point (edge). We start here too with the condition *true*, at the end of the path (i.e., after the last node). When we pass over a diamond node (on our way back), we either conjoin it as is, or conjoin its negation, depending on whether we exited this node 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 $\varphi[e/x]$.

Calculating the path condition for the example in Fig. 2 backwards, we start at the end of the path, i.e., point **D**, with a path condition *true*. Moving backwards through the assignment $y := y * 2$ to point **C**, we substitute every occurrence of y with $y*2$. However, there are no such occurrences in *true*, so the accumulated path condition remains *true*. Progressing backwards to point **B**, we now conjoin the negation of the condition $x > y$ (since point **C** was on the *no* outgoing edge of the condition), obtaining $\neg(x > y)$. This is now the condition to execute the path from **B** to **D**. Passing further back to point **A**, we have to relativize the accumulated path condition $\neg(x > y)$ with respect to the assignment $x := x + 1$, which means replacing the occurrence of x with $x + 1$, obtaining the same path condition as in the forward calculation, $\neg(x + 1 > y)$.

The choice of direction for calculating the path condition is affected by the direction in which the path is obtained. Accordingly, we may 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 as mentioned in Sect. 2.1, we may benefit from discarding a prefix (or a suffix, respectively) 'on-the-fly'.

2.3. Translating the specification

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

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

Program counter predicate, of the form *at l*, where l is a program counter label. If there are several processes, we may need to disambiguate this kind of predicate by mentioning also the process name, e.g., P_3 *at l*. Such a predicate holds in a state if the program counter is at the location whose label is mentioned, i.e., on the edge *entering* a node with the mentioned label.

Program variables assertion. A predicate that includes the program variables (and does not include further Boolean operators). Such a predicate is interpreted over a state according to the usual first order semantics.

These formulas may be combined using Boolean and temporal operators. Our implementation uses the linear temporal logic (LTL) syntax, 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. For a propositional sequence σ over $2^{\mathcal{P}}$, we denote the ith state (where the first state is numbered 0) by $\sigma(i)$, and the 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 over finite sequences 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 $0 \leqslant j < |\sigma|$ so that for each $0 \leqslant i < j$, $\sigma^{(i)} \models \varphi$.
- $\sigma \models \neg\varphi$ 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 temporal operators can be defined using the above operators. In particular, $\overline{\bigcirc}\varphi = \neg \bigcirc \neg\varphi$, $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$, $\varphi\, \mathcal{V}\, \psi = \neg((\neg\varphi)\, \mathcal{U}\, (\neg\psi))$, $true = p \vee \neg p$, $false = p \wedge \neg p$, $\Box\varphi = false\, \mathcal{V}\, \varphi$, and $\Diamond\varphi = true\, \mathcal{U}\, \varphi$. 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. 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 $\overline{\bigcirc}$*false* holds in a state that is in deadlock or termination.

The specification is translated into a finite state automaton. The algorithm we use is the one described in [GPVW95], and adapted to deal with *finite sequences*, as in [GP02], with further optimizations to reduce the number of nodes generated. Let (S, Δ, I, F, L) be a finite state automaton with nodes (states) S, a transition

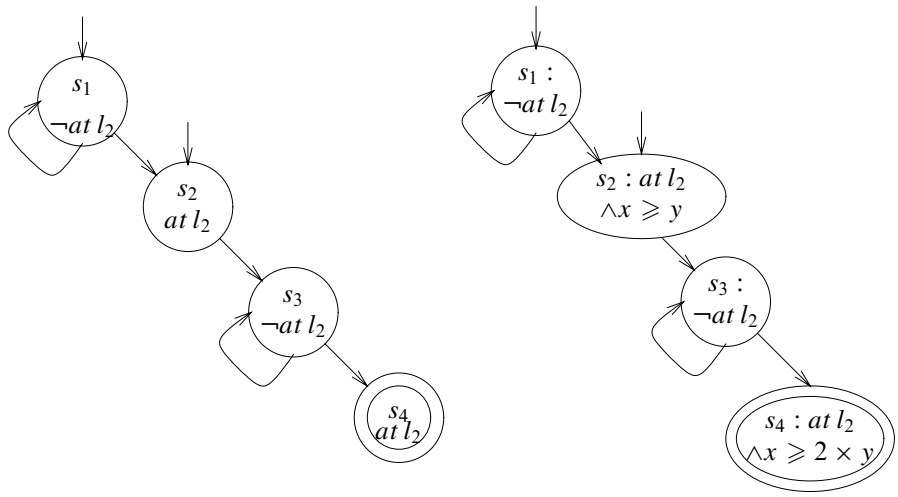

Fig. 3. Property automata

relation $\Delta \subseteq S \times S$, initial nodes $I \subseteq S$, accepting nodes $F \subseteq S$ and a labeling function L from S to some set of labels. A *run* of the automaton is a finite sequence of nodes $s_1 s_2 \ldots s_n$, where $s_1 \in I$, and for each $1 \leqslant i < n$, $(s_i, s_{i+1}) \in \Delta$. An *accepting* run satisfies further that $s_n \in F$.

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 negated or non-negated 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 nodes are accepting, hence $F^B = S^B$). Each node in S^B is labeled by (1) a single program counter value, (2) a node shape, e.g., box or a diamond, respectively), and (3) an assignment or a condition, respectively.

The *intersection* between a property automaton A and a flow chart B is an automaton $A \times B = (S^{A\times B}, \Delta^{A\times B}, I^{A\times B}, F^{A\times B}, L^{A\times B})$.

- The nodes $S^{A\times B} \subseteq S^A \times S^B$ have matching labels: if $(a, b) \in S^{A\times B}$ then the program counter of the flow chart node b must satisfy the program counter predicates labeling the property automaton node a.
- The transitions are $\{((a, b), (a', b'))|(a, a') \in \Delta^A \wedge (b, b') \in \Delta^B\} \cap (S^{A\times B} \times S^{A\times B})$.
- The initial states are $I^{A\times B} = (I^A \times I^B) \cap S^{A\times B}$.
- The accepting states are $F^{A\times B} = (F^A \times S^B) \cap S^{A\times B}$. Thus, membership in $F^{A\times B}$ depends only on the A automaton component being accepting.
- The label on a matched pair (a, b) in the intersection contains the separate labels of a and b.

Note that acceptance of runs by the intersection automaton are independent of the program variable assertions on the nodes S^A and the assignments and conditions labeling the nodes S^B. The right automaton in Fig.3 includes both program counter predicates and program variables assertions. When removing the program variables assertions, we obtain the automaton on the left of the same figure. The initial nodes are denoted with an incoming edge without a source node. The accepting nodes are denoted with a double circle.

We assume that each node $s \in S^A$ of the property automaton is annotated by some set of program variables assertions whose conjunction is η_s and some set of program counter predicates whose conjunction is μ_s. This annotation is generated automatically when translating an LTL formula into an automaton.

Now consider an accepting sequence of the intersection of the property automaton A and the flow chart B of the form $\tau = t_1, t_2, \ldots, t_n$. Projecting τ over the components of the flow chart gives a path. Thus, we may observe τ as a path with some assertions added to it.

We are interested in the set of execution sequences

$$\mathcal{S}(\tau) = \{\rho = g_1, g_2, \ldots, g_n | \rho \in exec(\tau) \wedge \forall i\ 1 \leqslant i \leqslant n, g_i \models \eta_{t_i} \wedge \mu_{t_i}\}$$

That is, executions of the path τ that also satisfy the corresponding temporal property expressed using the automaton A. (If A is constructed as a translation of an LTL property ψ then $\rho \models \psi$.)

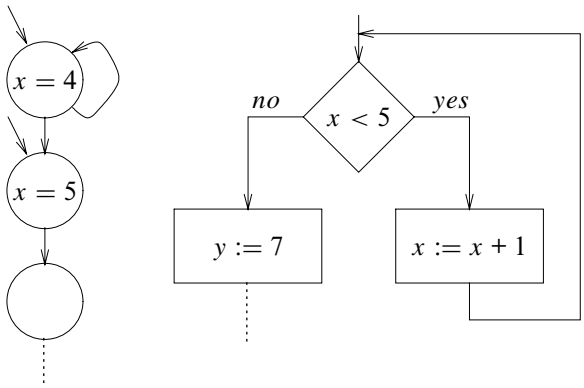

Fig. 4. Directing a search with an LTL property

We are interested in finding a condition for executing a path τ while satisfying a temporal property ψ. We denote such a condition by $\Gamma_{\tau,\psi}$. Any member of $\mathcal{S}(\tau)$ starts with some state satisfying $\Gamma_{\tau,\psi}$. Conversely, every state satisfying $\Gamma_{\tau,\psi}$ can be extended to an execution of τ. Again, for deterministic code, every state can be extended for at most one execution sequence. In this case, $\Gamma_{\tau,\psi}$ is satisfied *exactly* by states that start sequences in $\mathcal{S}(\tau)$.

Using a temporal specification on program counters

The proposed use for a temporal formula is to assist a human tester in inspecting suspicion paths and executions. For example, the executions of paths that passes through label l_2 twice may be suspicious of leading to some incorrect use of resources. The tester may express such paths in LTL as

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

Recall that the interpretation of LTL formulas is over finite paths, and we are looking for finite prefixes that satisfy the property. It is possible that such a path can be extended to pass through l_2 more than twice. This formula can be translated to the property automaton that appears on the left in Fig. 3.

Note that since the above temporal specification ψ involves only the program counters but not the program variables, for each path ρ there can be only two cases:

- All the executions of $exec(\rho)$ satisfy ψ, or
- None of the executions of $exec(\rho)$ satisfy ψ.

In the former case, $\bigcirc_{\tau,\psi} = \bigcirc_{\tau}$ and in the latter case $\bigcirc_{\tau,\psi} = false$. In this case, by taking the intersection of the property automaton A for ψ and the flow chart B, the paths that are the runs of the intersection are exactly those that have all of their executions satisfying ψ .

In symbolic execution, we are often incapable of comparing states, consequently, we cannot check whether we reach the same state again. We may not assume that two nodes in the flow chart with the same program counter labels are the same, as they may differ because of the values of the program variables. We also may not assume that they are different, since the values of the program variables may be the same. The flow chart and property automaton in Fig. 5 demonstrate why this lack of ability to identify states is problematic. Suppose that after passing the label l_2 the search always gives priority to label l_3, rather than to l_1. Note that both l_2 and l_3 satisfy $\neg at\, l_1$, hence match the top node of the property automaton, on the right in Fig. 5. If we consider each occurrence of a flow chart node as visited with a *different state*, the search will only progress repeatedly through the loop containing l_2 and l_3. It is also not a valid solution to assume that when flow chart nodes appear in the path multiple number of times, they appear with the *same state*. If we make this assumption, the path $l_2 l_3 l_2 l_3 l_2 l_1$, which contains the nodes l_2 and l_3 more than once, and which might be the one with the error, will not be generated during the search.

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

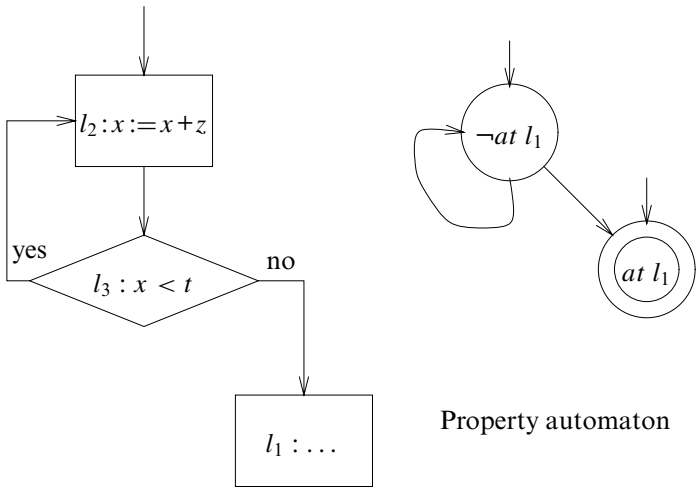

Fig. 5. A flow chart and a property automaton

counts the number of times this state has 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 can cover every finite path.

By not identifying when states are the same, we may run into a combinatorial explosion. For example, consider the state space in Fig. 6. 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 that can be explored is exponential with the number of diamonds. Such a graph can be automatically generated from a sequence of if-then-else statements, such as

if $x > y$ = then $x := x - y$
else $y := y - x$;
if $z > w$ then $z := z - w$
else $w := w - z$;
if $s > t$ then $s := s - t$
else $t := t - s$;

A practical remedy for this is to strengthen the LTL specification by mentioning more program counter predicates, so that most of the multiple choices will not be present in the intersection due to mismatch between the node labels and these predicates.

Taking into account the program variables assertions

The specification formula (1) is based only on the program counters. Suppose that we also want to express that when we are at the label l_2 for the first time, the value of x is greater or equal to the value of y, and that when we are at the label l_2 the second time, x is at least twice as big as y. We can write the specification as follows:

$$\psi = (\neg at\, l_2)\mathcal{U}(at\, l_2 \wedge x \geqslant y \wedge \bigcirc((\neg at\, l_2) \wedge ((\neg at\, l_2)\mathcal{U}(at\, l_2 \wedge x \geqslant 2 \times y)))) \tag{2}$$

An automaton obtained by the translation appears on the right in Fig. 3. The translation from a temporal formula to an automaton results in the program variables assertions $x \geqslant y$ and $x \geqslant 2 \times y$ labeling the second and fourth nodes. They do not participate in the automata intersection, but we need to incorporate them when calculating the path condition $\Gamma_{\tau,\psi}$. According to the (rather technical) definition of the automata intersection, the conjunction of the program variables assertions labeling the property automaton nodes are assumed to hold in the path condition *before* the effect of the matching flow chart node. Accordingly, if we add a condition η from some property automaton node to an assignment node in the flow chart, the assignment will take effect right after η has to hold.

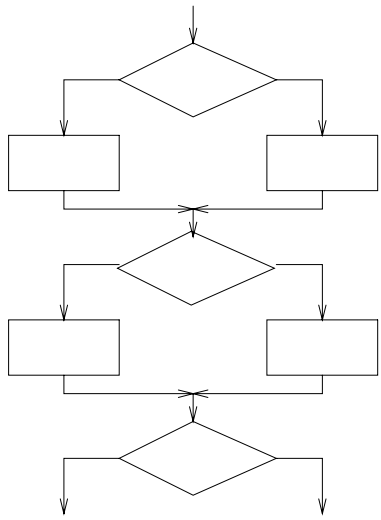

Fig. 6. An example with exponentially many sequences

When we intersect the property automaton in Fig. 3 with the flow chart on the left in Fig. 5, we may obtain a path with flow chart nodes $l_2 l_3 l_2$. Calculating the path condition for the original property (1) backwards starts with initially setting the accumulated path condition φ to *true*. We progress backwards over nodes l_2, l_3 and then l_2 again. Because of the assignment in the node labeled l_2 (which is the last node in the path), we relativize the accumulated path condition with respect to this assignment. But since it does not contain any variables, φ remains *true*. We now add now to the path condition as a conjunct the condition in l_3, obtaining $x < t$. Again because of the node labeled l_2 (this time its first occurrence in the path), we relativize φ with respect to the assignment $x := x + z$, and obtain $x + z < t$.

In general, when calculating a path condition for a path τ obtained from the intersection of the property automaton for a property ψ and a flow chart, we need to take into account program variables assertions that appear on it (coming from the property automaton components). We can do that by transforming the path as follows. Observe that each node in the intersection is a pair (a, b), where a is a property automaton node, and b is a flow chart node in the current path. The label of b agrees with the program counter predicates in a. Otherwise, the path is automatically rejected to be in the intersection during the search (and $\Gamma_{\tau,\psi} = \mathit{false}$). We transform each such pair into two sequential nodes. First, b remains as it appears in the flow chart. We insert a new diamond node to the current path, just before b. The inserted node contains as its condition the conjunction of the program variables assertions labeling the node a (and *true* if there are no program variables assertions labeling a). The edge between the new diamond and b is labeled with 'yes' corresponding to the case where the condition in a holds. The edge that was formerly entering b now enters the new diamond. A standard induction on the length of the path shows that the path condition $\Gamma_{\tau'}$ for the path τ' obtained by the transformation is exactly $\Gamma_{\tau,\psi}$.

We return now to the property (2), whose corresponding property automaton appears on the right in Fig. 3. Again, we can obtain a path containing the flow chart nodes $l_2 l_3 l_2$ in that order. In fact, the path is $(s_2, l_2)(s_3, l_3)(s_4, l_2)$. We insert three new diamonds corresponding to the nodes s_2, s_3 and s_4, as shown in Fig. 7. Thus, the order according to which we pass through the nodes on the way backwards is l_2, s_4, l_3, s_3, l_2 and finally s_2. According to the labeling on the property automaton components, the condition $x \geqslant 2 \times y$ labels s_4, and $x \geqslant y$ labels s_2 (due to lack of program variables assertions, the node s_3 is labeled with *true*). Again, calculating the path condition backwards, we start with *true*. Relativizing φ with respect to the assignment in l_2 retains *true*. At this point, we add the conjunct $x \geqslant 2 \times y$ due to s_4, obtaining $\varphi = x \geqslant 2 \times y$. We reach the node labeled with l_3, and add the conjunct $x < t$, obtaining $x \geqslant 2 \times y \wedge x < t$. The node s_3 does not contribute anything to the path condition. We now reach node l_2, relativizing again with respect to the assignment $x := x + z$, we obtain $x + z \geqslant 2 \times y \wedge x + z < t$. At this point we also need to add the conjunct $x \geqslant y$, labeling s_2, obtaining the condition $x + z \geqslant 2 \times y \wedge x + z < t \wedge x \geqslant y$.

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. Our separation of the search depends on the fact that we do

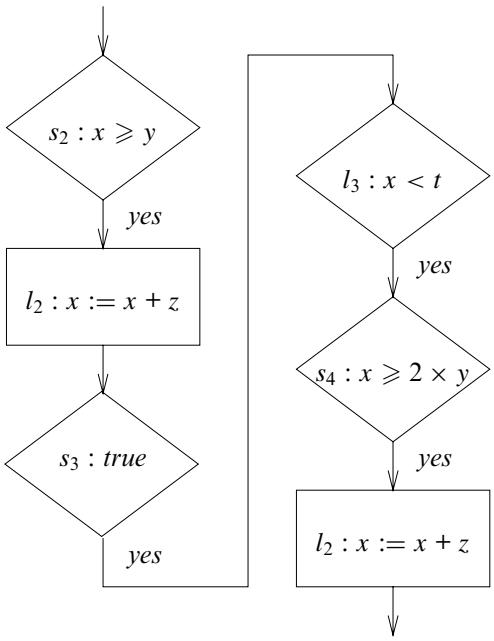

Fig. 7. A a combined path due to intersection

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 [MP91a, MP91b], and can usually be translated (unfortunately with some increase to the size of the formula) into formulas that make the required separation.

We illustrate how the LTL property can direct the search in Fig. 4. The LTL formula is $x = 4\,\mathcal{U}\,(x = 5 \wedge \bigcirc \ldots)$. (This is an incomplete formula, we consider only the prefix of the specification, that has to start with some states satisfying $x = 4$, then have $x = 5$ and ignore what happens further.) In this case, by incrementally calculating the path conditions for prefixes of the search path, we can (in this case) rule out and immediately backtrack from some directions in the search. An (incomplete) automaton for this property appears on the left hand side of the figure. When this is intersected with the portion of code (again incomplete) on the left, the search can proceed in the the following way:

- First, the automaton node marked $x = 4$ is matched with the condition flow chart node $x < 5$.
- The left edge marked with *false* cannot be used for this search since then the path condition to pass a condition node with $\neg x < 5$ when $x = 4$ holds is contradictory, hence the path condition is *false*. Hence, an attempt to extend the previous path by matching the node $x = 4$ (or $x = 5$) with the assignment node $y := 7$ would result an immediate backtrack, and the right edge, marked *true*, must be selected.
- We cannot extend the path by pairing up the node $x = 5$ with the node $x := x + 1$. This is because $x = 5$ *before* $x := x + 1$ and hence *after* the condition node $x < 5$, with $x = 4$ *before* that node and no assignment in between, would result in again a *false* path condition.
- We pair up the node $x = 4$ with the node $x := x + 1$.
- We now try to pair up $x = 4$ with a second occurrence of $x < 5$. The path condition calculates to *false* since the node $x := x + 1$ has $x = 4$ both before and after it. Thus, we need to pair the second occurrence of $x < 5$ with $x = 5$.
- Now we cannot take the right edge and match the second occurrence of $x := x + 1$ with any node from the property automaton. This is because the combination of $x = 5$ before the condition $x < 5$ and taking the condition exit edge labeled *true* is contradictory would result in a *false* path condition. Thus, we continue with the left exit edge labeled *false* and match the node $y := 7$ with the next automaton node and so forth.

2.4. Incorporating specifications for missing code

In 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 use a *specification* for drivers and stubs, instead of using their code.

Instead of using a stub, our method prescribes replacing a procedure with an assertion that relates the program variables *before* and *after* its execution. We call such assertions *stub specifications*. In a stub specification, the original program variables, e.g., x, y, represent the values of the variables just *before* the execution of the specified procedure. The primed version of the variables, e.g., x', y', represent the values *after* the execution. We also allow the predicate *same*, with a list of variables as its arguments, 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 $x' = x \wedge y' = y$.

Note that we treat the missing procedures as atomic, where their code is replaced by a specification of their input-output relation. The temporal specification should be given accordingly.

We can easily incorporate such a procedure call into our calculation of the path condition. We start with the (simpler) 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 stub specification formula, expressing the effect of the procedure, where V' is the list of variables V primed [5]. The accumulated path condition, which includes the effect of the procedure, becomes $\exists V'(\varphi[V'/V] \wedge \eta)$. (If $V = \{x, y\}$, this notation is a shorthand for $\exists x' \exists y'(\varphi[x'/x][y'/y] \wedge \eta)$.) That is, the formula obtained by conjoining η to a version of φ, in which 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 following considerations are needed for handling the annotations of the property automaton. Suppose we have calculated the accumulated path condition $\varphi(V)$ so far, 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, in the same way 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 $\mu(V)$, and the current path condition be $\varphi(V)$. Then the path condition becomes now $\mu \wedge \varphi$.

The forward calculation of the path condition is more complicated when stub specifications are used. We cannot keep the symbolic values of each variable since their change is given as a formula, i.e., a relation between states, instead of an assignment, i.e., a function of the current state. 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. Recall that 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 $\exists V' \varphi(V, V')$.

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

- A *diamond node*, with condition $\eta(V)$. We transform the accumulated path condition $\varphi(V, V')$ into $\varphi \wedge \eta[V'/V]$.
- A *stub* node with a condition $\eta(V, V')$. (We denote such a node with a double box.) The new path condition is $\exists U(\varphi[U/V'] \wedge \eta[U/V])$, with new variables U. That is, we unify, by renaming, the variables V' of φ and the variables V of η, and then eliminate them using existential quantification.
- *Assignment (box)* $x := e$. This can be treated in the same way as a stub specification with an assertion $x' = e$, using the *same* predicate with parameters that include all the program variables, except for x. Correspondingly, the resulted condition, after eliminating some existential quantifiers, becomes $\exists t\ (\varphi[t/x'] \wedge x' = e[t/x])$, 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, one from the property automaton and one from 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

[5] We use $\varphi(V)$ and φ interchangeably, with the former denoting that the free variables of the formula are among the list V, and the latter is used when V is already clear from the context. A similar consideration applies to $\eta(V, V')$ and η.

now $\mu[V'/V] \wedge \varphi$. That is, the variables in the condition μ are renamed to refer to the set of variables V', representing the values at the current point in the path, and the obtained condition is added to the accumulated path condition.

The forward calculation, in the presence of stubs specifications involves introducing new existential quantifiers. These quantifiers may considerably complicate the path condition, and we would like to eliminate them 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 $\Theta(V)$. Accordingly, if the checked temporal condition is μ, we check $\Theta \wedge \mu$. Effectively, this means that we conjoin each path condition with Θ, i.e., restrict the executions to those satisfying Θ initially.

3. An example

In this section we consider a larger example, whose code is given in an appendix, at the end of the paper. Consider the Euclid greatest common divisor (gcd) algorithm in Fig. 8. The initial condition for this procedure $\Theta = a > 0 \wedge b > 0 \wedge at\ l_0$. The expression $x\ rem\ y$ denotes the remainder of dividing x by y. At the end of the execution, the value of x should be the gcd of a and b. The algorithm uses a procedure to calculate the remainder 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 [GP02]. An error is introduced into this presentation, as the nodes labeled by l_4 and l_5 are reversed from the correct algorithm.

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. It 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 Fig. 9.

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, our system forms the path condition for the above path while adding for each node in the path the program variables assertions from the matching property automaton node. 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 $a > 0 \wedge b > 0 \wedge a\ rem\ b = 0$, 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 for this sequence, when simplified, is equivalent to $a > 0 \wedge b > 0 \wedge a\ rem\ b \neq 0$. This means that when b does not divide a we terminate with $x = 0$ after two traversals of the loop.

We can stop the search now, since $a > 0$ and $b > 0$ is part of the initial condition (and a and b never change), and it is always the case that either $a\ rem\ b = 0$ or $a\ rem\ b \neq 0$. Thus, we terminate with $x = 0$, either after one or two executions of 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. Since we have already covered all the possibilities, we will not find any more paths. There are paths in the intersection of the flow chart graph and the property automaton. 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. Implementation

Most features in this paper (with the exception of replacing stubs with assertions over the variables at the points of call and return) have been implemented as an extension to an existing system for temporal debugging, PET [GP02]. In that system, code for concurrent programs is translated into visual flow charts. The user may select a potential execution path by selecting consecutive nodes in the flow charts of the processes that constitute the concurrent program and the system will produce the path condition necessary for that path to be executed.

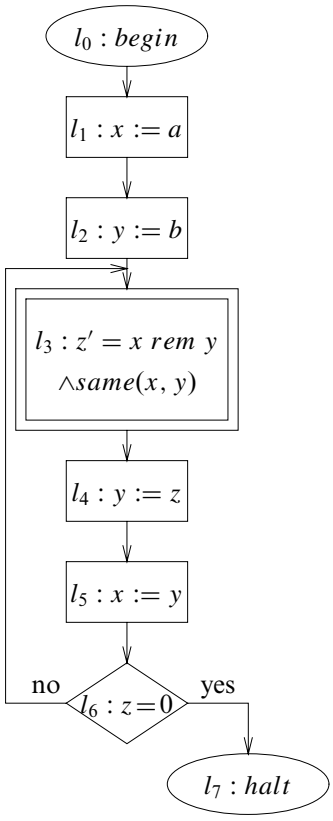

Fig. 8. Euclid's gcd algorithm with an error

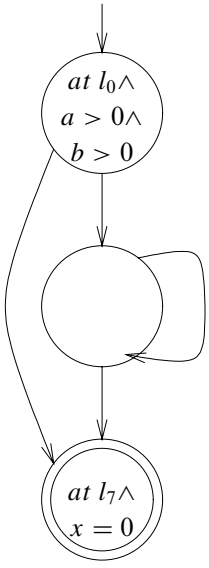

Fig. 9. A property automaton with two kinds of labels

Alternatively, the user may specify a linear temporal logic property and request either a minimal or maximal (acyclic) path satisfying the given property. The system searches the full state space of the concurrent program and if there is one, returns a path satisfying the desired property together with the most general condition allowing that path to execute (the path condition). The user has the option to ask the system to backtrack and look for another solution, to discard the search and start again, or to extend the given path with another search beginning where the previous search ended. In these searches we use the full state space consisting of all the states reachable starting from a state where all variables are initialized to 0. In the original version of PET, path conditions were calculated after the path was found, using the backward method described earlier in this paper.

The external manifestation of the extensions described in this paper is as an additional search option. One can still specify a property to hold of a path, but this time one may choose for the search to take place symbolically, using the product of the flow chart and property automaton instead of the full state space. The user is queried for the maximum number of times that a node may recur in a successful path. The result returned is again a path and a path condition.

The PET system comprises of two top-level pieces: a graphical user interface written in Tcl/Tk, and a back end computing engine written in SML [MTHM97]. In more recent versions we have incorporated the graphical user interface with the computing engine written as a single top-level program in SML, using the SML-Tk library [Lue99]. In addition to the main back end computing engine, PET makes use of two external tools, namely DOT [GN00] for calculating graphical layouts, and the theorem prover HOL [GM93] for additional simplification of terms.

The extension to PET described in this paper required the addition of two new modules, one for the calculation of symbolic environments and one for the creation of the finite state automaton corresponding to the product of the flow charts of a concurrent program. Additional modules were altered with the addition of new functionality and the alteration of existing ones.

5. A larger example

Consider the following problem: you wish to know, for a given initial complex valued value C, whether the sequence $C(i) = C(i-1)^2 - C(i-2)$ diverges to infinity, where $C(0) = 0$ and $C(1) = C$, and if it diverges, how fast. It is believed that any value whose real or imaginary parts are greater than 4 in absolute value diverge rapidly, and will be classified as having already reached the diverging threshold. Values that are quite close to 0 will fail to diverge, so it is the moderate sized values that are of the most interest. Also, there is reason to believe that values that are close together may have significantly different behavior. Therefore, it is desirable to maintain a fairly high degree of accuracy for those moderate values. For this reason, it has been decided that we will use an integer encoding of fixed-point reals, keeping the decimal point after the two highest bits for accuracy. During intermediate computation we will want to keep overflow information, since intermediate values may exceed the bounds for final values. We have implemented in the language of PET a procedure which, given a complex value C entered as a pair of integers `CReal` and `CIm` representing the corresponding fixed-point real and imaginary parts, returns the lowest number of iterations i such that the absolute value of either the real or imaginary part of $C(i)$ is greater than 4.

Although we consider the input to the procedure to be a pair of integers representing fixed-point reals, we will internally use a pair of integers for holding each of the fixed-point reals. Our representation is designed to keep 30 bits of accuracy with the decimal point after the first 28 bits. The lowest 15 bits of the number are kept in a nonnegative number. The high 15 bits of a number, together with any overflow, are kept in a signed integer of the same sign as the number. If (x_h, x_l) represents x then

$$x = x_h * 2^{15} + x_l.$$

The PET code for our implementation of this example may be found in the Appendix. The code has been augmented by a numbering of the program statements. This numbering is not a part of the actual code. These numbers are the node numbers in the flow chart generated by PET during the translation of the code. We do not include the image of the flow chart as it is too long to be meaningfully presented in the limited space afford by the page size of a paper publication. However, the flow chart is easily traversed using scroll bars on a computer, and in that setting we found its size was no problem.

We begin by initializing a counter variable `i`, four variables `CRealHigh`, `CRealLow`, `CImHigh` and `CImLow` for the high and low bits of C, and four more variables, `OldCRealHigh`, `OldCRealLow`, `OldCImHigh` and `OldCImLow` all holding 0, for $C(0) = 0$. After this initial setup, the program is a while loop which terminates either when the

overflow value of C is greater than 0 or less than -1, or when `i` exceeds a cutoff threshold of 1000. Inside the while loop we calculate $\mathtt{C}^2 - \mathtt{OldC}$, assigning the value of `C` to `OldC` and assigning this new value to `C`. There is a collection of additional local variables used for carrying out this computation.

In an initial version of this program, we accidentally had the following condition for the while loop:

```
60:  while (CRealHigh / power15 = 0 or CRealHigh / power15 = ~1) and
       (CImHigh / power15 = 0 or CImHigh = ~1) and i < 1000 do
```

This code has an error, in that `CImHigh` is not divided by `power15` before being compared to `~1`. If we suspect that the while loop is sometimes terminating too early, we can examine the path condition for the path from the start node (node # 0) directly to the end node (node # 1). With the incorrect condition in the while loop, this condition is calculated by PET as:

```
(CReal / 1073741824 =/= 0 and CReal / 1073741824 =/= ~1) or
(CIm / 1073741824 =/= 0 and CIm / 32768 =/= ~1)
```

>From this condition we can see that the lower bound for `CIm` is being handled differently than the other bounds. Once we have corrected this problem, the while loop condition becomes:

```
60:  while (CRealHigh / power15 = 0 or CRealHigh / power15 = ~1) and
       (CImHigh / power15 = 0 or CImHigh / power15 = ~1) and i < 1000 do
```

and the path condition for the straight path from the start node to the end node (not going through the loop) becomes

```
(CReal / 1073741824 =/= 0 and CReal / 1073741824 =/= ~1) or
(CIm / 1073741824 =/= 0 and CIm / 1073741824 =/= ~1)
```

and we see that the asymmetry has been resolved.

The calculations for $\mathtt{C}^2 - \mathtt{OldC}$ are fairly involved and are subject to error. Our tool can be used to gain confidence in this part of the program. We would like to be able to know, for example, that `CRealLow` and `CImLow` always are nonnegative but less than 2^{15} at the intial entry of the while loop, and that

$$\mathtt{CReal} = \mathtt{CRealHigh} * 2^{15} + \mathtt{CRealLow}$$

and

$$\mathtt{CIm} = \mathtt{CImHigh} * 2^{15} + \mathtt{CImLow}.$$

This can be checked by searching for a path satisfying the LTL formula

$$\begin{array}{l}(\neg at\, l_{while})\mathcal{U}(at\, l_{while} \wedge \\ (\mathtt{CRealLow} < 0 \vee \mathtt{CRealLow} \geqslant 2^{15} \vee \\ \mathtt{CImLow} < 0 \vee \mathtt{CImLow} \geqslant 2^{15} \vee \\ (\neg(\mathtt{CReal} = \mathtt{CRealHigh} * 2^{15} + \mathtt{CRealLow})) \vee \\ (\neg(\mathtt{CIm} = \mathtt{CImHigh} * 2^{15} + \mathtt{CImLow}))))\end{array}$$

where l_{while} is the label for the node at the start of the while loop (node 60). Note that this is a search in the negative, that is a search for a counter-example and we wish to be told that no path satisfying this condition exists. In practice, it would generally be best to break a conjunct such this (or larger) up into separate smaller formulae. Splitting the fixed-point reals into the separate pieces involves using integer division (a built-in primitive in PET). Prior to simplification, the path condition generated for these formulae involve expressions involving this division. The ability to eliminate the existence of a path satisfying any of the LTL properties is heavily dependent on the strength of the auxiliary simplification engine from HOL. In this case, HOL did have the strength to simplify these expressions to false, but it falls outside the realm of Presburger arithmetic. The condition as returned from PET using only its built-in simplifications without further assistance from HOL is

```
((CReal - ((CReal / 32768) * 32768) >= 32768 or
CIm - ((CIm / 32768) * 32768) >= 32768) or
CReal =/=
((CReal / 32768) * 32768) + (CReal - ((CReal / 32768) * 32768))) or
CIm =/= ((CIm / 32768) * 32768) + (CIm - ((CIm / 32768) * 32768))
```

To further simplify this, HOL uses the conditional inequalities

$$y > 0 \Rightarrow x - ((x/y) * y) \geqslant 0$$

and

$$y > 0 \Rightarrow x - ((x/y) * y) < y$$

together with the rewrite

$$(a \geqslant b) = not(a < b).$$

In this case, because we are always dividing by constants, HOL can always decide that the divisor is strictly greater than 0. Were we to have complex expressions in the divisor, we could easily generate expressions that HOL would fail to simplify.

Once we have established that the values that are passed into the while loop the first time are correct, we would still like at least some assurance that the later values are also plausible. For this we can use LTL formulae such as

$$\Diamond(at\, l_{while} \wedge (\neg \texttt{CRealLow} \geqslant 0))$$

and

$$\Diamond(at\, l_{while} \wedge (\neg \texttt{CRealLow} > 2^{15}))$$

to seek possible error cases. Since we have already verified that the initial input to the while loop is split correctly, we also know that all possible legitimate values can be input. Therefore, to seek errors in the inner algorithm of the while loop, it suffices to limit the number of times that the sought path can pass through the while loop to one. Again, for each such condition, the simplification engine was able to reduce the path condition after a single pass through the while loop to false. This increases our confidence in our code, at least with respect to these simple cases. It should be noted that, while we have increased our confidence in the code, it is still possible that we have swapped the roles of some variables or made other mistakes in the code.

The PET system is run in the interactive compiler SML of NJ. The Higher-Order Logic theorem prover HOL90 is run in SML as a library accessed by PET. Within this system, the time to parse and construct internal data structures for the PET program given in the Appendix was 11μ-seconds. The time to verify that no path satisfied

$$\begin{aligned}&(\neg at\, l_{while})\mathcal{U}(at\, l_{while} \wedge \\ &(\texttt{CRealLow} < 0 \vee \texttt{CRealLow} \geqslant 2^{15} \vee \\ &\texttt{CImLow} < 0 \vee \texttt{CImLow} \geqslant 2^{15} \vee \\ &(\neg(\texttt{CReal} = \texttt{CRealHigh} * 2^{15} + \texttt{CRealLow})) \vee \\ &(\neg(\texttt{CIm} = \texttt{CImHigh} * 2^{15} + \texttt{CImLow}))))\end{aligned}$$

took 2.906 milliseconds, and the time to verify that no path (passing through the while node at most once) satisfies

$$\begin{aligned}\Diamond(at\, l_{while} \wedge ((\neg \texttt{CRealLow} \geqslant 0) \vee (\neg \texttt{CRealLow} > 2^{15}) \vee \\ (\neg \texttt{CImLow} \geqslant 0) \vee (\neg \texttt{CImLow} > 2^{15})))\end{aligned}$$

took 9.625 milliseconds. These times were taken using the Standard ML standard library Timer, using the functions `startRealTimer` and `checkRealTimer`. These times do not include the time to parse the LTL formulae. These measurements where made on an Apple PowerBook with a 1.33 GHz PowerPC G4 processor and 2 GB DDR SDRAM memory, running Mac OS X version 10.3.8.

6. Discussion

We proposed and implemented a symbolic verification approach for a unit of code, which we call unit checking. It allows verifying a piece of code in isolation (provided some specification of the other pieces is given), akin to unit testing. It results in a collection of path conditions, which can be used to derive tests of the verified code. Our approach was described so far for the verification of (linear) 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 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 [Mat93]. It is decidable for certain specific domain, e.g., for finite domains and for Presburger Arithmetic. We apply various procedures for simplifying formulas, and for refuting Presburger arithmetic assertions [Opp78] (we apply it repeatedly to subformulas). Because of 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 minimal length path in the intersection of the automata. Moreover, there can be infinitely many paths in the intersection.

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 for the translated code, and adapting the path condition calculation to these constructs. Instead of flow chart nodes, we deal with tuples of such node, obtained through the Cartesian product of flow charts. Orthogonal work generalizes the path condition analysis to handle also pointers and aliasing [VG02].

Our framework can also be used with concurrent code. In fact, our implementation supports concurrency. In a concurrent program, there are several flow charts that interact with each other. A path in a concurrent system is an interleaved sequence of nodes, whose projections on each flow chart for a process is a sequential path. However, care should be taken, as the interaction between the variables of procedures represented by stub specifications can cause additional behaviors that are not present in our analysis and invalidate the atomicity of the stubs. Moreover, concurrency introduces nondeterminism. In the presence of nondeterminism, executing the code from a state satisfying a path condition for a given concurrent (i.e., interleaved) path *does not* guarantee repeating the execution according to the path condition, rather *allows* it [GP02]. That is, in the presence of a nondeterministic choice, from each state satisfying the path condition, there *exists* an execution which follows the path for which the condition was calculated. In case of concurrency, a different part of the system, which transforms the code to force the execution of a selected test case, can be invoked [PQ04].

Our approach can be used for automating the unit testing process. Accordingly, a tester uses a formula to focus on some suspicious executions. 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. Because of the decomposition of the search according to the program counters and according to the program variables, our search is most efficient when the temporal specification is mainly (but not necessarily completely) dependent on the path (i.e., the program counters values) traversed during the execution (e.g., visits some loop three or four times).

The success of calculating the path condition and simplifying it into a form that is intuitive to the user is essential in our work. We currently used our own heuristics, as well as a decision procedure for Presburger arithmetic [Opp78] implemented within the theorem prover HOL. Another possibility is to use the Omega library [PW95]. It is interesting to observe that theorem provers include a lot of knowledge about simplifying arithmetical expressions. An open environment for a theorem prover, giving access to the part that deals with simplification, is very useful. Two theorem provers that make heavy use of simplifications are ACL2 [KMM00] and PVS [ROS98].

References

[BGP99] Bultan T, Gerber R, Pugh W (1999) Model-checking concurrent systems with unbounded integer variables: symbolic representation, and experimental results, ACM Trans Program Syst 21:747–789

[CGP00] Clarke EM, Grumberg O, Peled D (2000) Model checking, MIT Press

[Dij75] Dijkstra EW (1975) Guarded commands, nondeterminacy and formal derivation of programs. Commun ACM 18(8):453–457

[FLLN02] Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R (2002) Extended static checking for Java, PLDI pp 234–245

[GN00] Gansner ER, North SC (2000) An open graph visualization system and its applications to software engineering. Softw Pract Exp 30:1203–1233

[GP02] Gunter EL, Peled D (2002) Temporal debugging for concurrent systems, TACAS 2002, Grenoble, France, LNCS 2280. Springer, Berlin Heidelberg New York, pp 431–444

[GPVW95] Gerth R, Peled D, Vardi MY, Wolper P (1995) Simple on-the-fly automatic verification of linear temporal logic PSTV95, Protocol Specification Testing and Verification, Chapman & Hall pp 3–18

[GL80] Gries D, Levin G (1980) Assignment and procedure call proof rules, ACM Trans Program Lang Sys 2:564–579

[GM93] Gordon MJC, Melham T (1993) Introduction to HOL. Cambridge University Press

[GLSU02] Hong HS, Lee I, Sokolsky O, Ural H (2002) A temporal logic based theory of test coverage and generation, tools and algorithms for the construction and analysis of systems, Grenoble, France, LNCS 2280, pp 327–341

[KPV01] Kesten Y, Pnueli A, Vardi MY (2001) Verification by augmented abstraction: the automata-theoretic view, JCSS 62:668–690

[KMM00] Kaufmann M, Manolios P, Moore JS (2000) Computer-aided reasoning: an approach, Kluwer Dordrecht

[King96] King JC (1976) Symbolic execution and program testing. Commun ACM 17(7):385–395

[Lue99] Lüth C, Wolff B (1999) Functional design and implementation of graphical user interfaces for theorem provers. J Funct Program 9(2):167–189

[Mat93] Matiyasevich Y (1993) Hilbert's tenth problem. MIT Press

[Mye79] Myers GJ (1979) The art of software testing. Wiley, New York

[MP91a] Manna Z, Pnueli A (1991) Completing the temporal picture. Theor Comput Sci 83:97–130

[MP91b] Manna Z, Pnueli A (1991) The temporal logic of reactive and concurrent systems: specification Springer, Berlin Heidelberg New York

[MTHM97] Milner R, Tofte M, Harper R, MacQueen D (1997) The definition of standard ML (revised). MIT Press, Cambridge

[Opp78] Oppen DC (1978) A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. JCSS 16(3):323–332

[Pel02] Peled D, (2002) Software reliability methods. Springer, Berlin Heidelberg New York

[PQ04] Peled D, Qu H (2004) Enforcing concurrent behaviors, accepted to RV'2004, Runtime Verification, Barcelona, 2004, to appear in ENTCS

[RP85] Rapps S, Weyuker EJ (1985) Selecting software test data using data flow information. IEEE Trans softw Eng SE-11 4:367–375

[ROS98] Rushby JM, Owre S, Shankar N (1998) Subtypes for specifications: predicate subtyping in PVS. Trans Softw Eng 24(9):709–720

[PW95] Pugh W, Wonnacott D (1995) Going beyond integer programming with the omega test to eliminate false data dependencies. IEEE Trans Parallel Distrib Sys 6:204–211

[VG02] Visvanathan S, Gupta N (2002) Generating test data for functions with point input 17th IEEE international conference on automated software engineering. Edinburgh, Scotland pp 149–162

Appendix The code for the example in Sect. 3

```
0:  begin
1:   power15 := 2 ^ 15;
2:   power14 := 2 ^ 14;
3:   power13 := 2 ^ 13;
4:   power12 := 2 ^ 12;

5:   i := 0;

6:   CRealHigh := CReal / power15;
7:   CRealLow := CReal - (CRealHigh * power15);

8:   CImHigh := CIm / power15;
9:   CImLow := CIm - (CImHigh * power15);

10:  OldCRealHigh := 0;
11:  OldCRealLow := 0;

12:  OldCImHigh := 0;
13:  OldCImLow := 0;

60:  while
      (CRealHigh / power15 = 0 or CRealHigh / power15 = ~1) and
      (CImHigh / power15 = 0 or CImHigh / power15 = ~1) and
      i < 1000
     do
```

```
     begin
14:   i := i + 1;

15:   CImLowSquare := CImLow * CImLow;
16:   CImLowSquareCarry := CImLowSquare / power15;
17:   CImLowSquareBase :=
       CImLowSquare - (CImLowSquareCarry * power15);

18:   CRealLowSquare := (CRealLow * CRealLow) - CImLowSquareBase;

19:   CLowSquareDiffCarry :=
      (CRealLowSquare / power15) - CImLowSquareCarry;

20:   CIm2HighLow := (CImHigh * CImLow);
21:   CIm2HighLowCarry := CIm2HighLow / power14;
22:   CIm2HighLowBase :=
       (CIm2HighLow - (CIm2HighLowCarry * power14)) * 2;

23:   MiddleCarry := CLowSquareDiffCarry - CIm2HighLowBase;
24:   MiddleCarryHighBits := MiddleCarry / 2;
25:   MiddleCarryLowBit := MiddleCarry - (MiddleCarryHighBits * 2);

26:   CRealHighLow := (CRealHigh * CRealLow) + MiddleCarryHighBits;
27:   CReal2HighLowCarry := CRealHighLow / power14;
28:   C2HighLowDiffBase :=
       (CRealHighLow - (CReal2HighLowCarry * power14)) * 2;

29:   CImHighSquare := (CImHigh * CImHigh);
30:   CImHighSquareCarry := CImHighSquare / power15;
31:   CImHighSquareBase :=
       CImHighSquare - (CImHighSquareCarry * power15);

32:   CRealHighSquare :=
       ((CRealHigh * CRealHigh) +
        CReal2HighLowCarry - CIm2HighLowCarry) - CImHighSquareBase;

33:   CRealHighSquareCarry := CRealHighSquare / power15;
34:   CHighSquareDiffBase :=
       CRealHighSquare - (CRealHighSquareCarry * power15);

35:   CHighSquareDiffCarry :=
       CRealHighSquareCarry - CImHighSquareCarry;

36:   CRealImLow := CRealLow * CImLow;
37:   CRealImLowCarry := CRealImLow / power15;

38:   CRealHighImLow := (CRealHigh * CImLow) + CRealImLowCarry;
39:   CRealHighImLowCarry := CRealHighImLow / power15;
40:   CRealHighImLowBase :=
       CRealHighImLow - (CRealHighImLowCarry * power15);

41:   CRealLowImHigh := (CRealLow * CImHigh) + CRealHighImLowBase;
42:   CRealLowImHighCarry := CRealLowImHigh / power15;
43:   CRealImMidBase :=
       CRealLowImHigh - (CRealLowImHighCarry * power15);
```

```
44:   CRealHighImHigh :=
      (CRealHigh * CImHigh) + CRealHighImLowCarry +
      CRealLowImHighCarry;

45:   CHighSquareDiffBaseTopBits := CHighSquareDiffBase / power13;

46:   NewCRealLow :=
      ((CHighSquareDiffBase -
        (CHighSquareDiffBaseTopBits * power13)) * 4) +
       (C2HighLowDiffBase / power13) - OldCRealLow;

47:   NewCRealLowCarry := NewCRealLow / power15;

48:   NewCRealHigh :=
       (CHighSquareDiffCarry * 4) + CHighSquareDiffBaseTopBits +
       NewCRealLowCarry - OldCRealHigh;

49:   NewCImLow := ((CRealHighImHigh - (CImHigh * power12)) * 8) +
                    (CRealImMidBase / power12) - OldCImLow;

50:   NewCImLowCarry := NewCImLow / power15;

51:   NewCImHigh :=
       (CRealHighImHigh / power12) + NewCImLowCarry - OldCImHigh;

52:   OldCRealLow := CRealLow;
53:   CRealLow := NewCRealLow - (NewCRealLowCarry * power15);

54:   OldCRealHigh := CRealHigh;
55:   CRealHigh := NewCRealHigh;

56:   OldCImLow := CImLow;
57:   CImLow := NewCImLow - (NewCImLowCarry * power15);

58:  OldCImHigh := CImHigh;
59:  CImHigh := NewCImHigh

     end

61: end.
```

Received February 2004
Revised September 2004 and April 2005
Accepted April 2005 by M. Leuschel and D. J. Cooke
Published online 5 August 2005