Software Analysis and Model Checking

Gerard J. Holzmann

Bell Laboratories, Lucent Technologies,
Murray Hill, New Jersey 07974, USA.
gerard@research.bell-labs.com

Abstract. Most software developers today rely on only a small number of techniques to check their code for defects: peer review, code walkthroughs, and testing. Despite a rich literature on these subjects, the results often leave much to be desired. The current software testing process consumes a significant fraction of the overall resources in industrial software development, yet it cannot promise zero-defect code. There is reason to hope that the process can be improved. A range of tools and techniques has become available in the last few years that can asses the quality of code with considerably more rigor than before, and often also with more ease. Many of the new tools can be understood as applications of automata theory, and can readily be combined with logic model checking techniques.

1 Introduction

Humans occasionally make mistakes, even programmers do. Even though mistakes are generally unpredictable, within fixed domain we can often predict fairly accurately just how many mistakes will be made. For programmers in industrial software development, the residual software defect ratio (the number of latent faults that remain in the code at the end of the development process) is normally somewhere between 0.5 and 5 defects per one thousand lines of non-comment source code [H01]. Curiously, this ratio is not unique to programming.

The New York Times appears seven times a week, with an average of 138,000 words per weekday issue, and 317,000 per Sunday issue [AM02]. Despite great care in fact checking, proof reading and spell checking, inevitably seven times a week a list appears of the most important mistakes that made it into print on the previous day. On average, the list of corrections contains 10 entries per day. At an average of ten words per sentence, this gives a fairly predictable residual defect density of one defect per one thousand sentences written.

So without knowing anything about the particulars of a given industrially produced software product, one thing is generally safe to assume: it has bugs. The same is of course true for all industrial products, but what makes the problem unique in software is that the effects of even very minor programming mistakes can cause major system failures. It is very hard to contain the potential effects of a software defect, especially in distributed systems software.

We will not argue in this paper that traditional software testing, peer reviews and code walkthroughs can or should be replaced wholesale. Most studies agree that this combination of techniques effectively catches the majority of design and coding errors. Yet, undeniably at the end of this process the residual software defect density ratio is not zero. Even at a low residual defect density of 0.1 defect per one thousand lines of

```
do {
        lock( &devExt->writeListLock );
        nPacketsOld = nPackets;
        request = devExt->WriteListHeadVa;
        if (request && request->status)
        {       devExt->WriteListHeadVa = request->nxt;
                unlock(&devExt->writeListLock);
                /* ... */
                nPackets++;
        }
} while (nPackets != nPacketsOld);
unlock(&devExt->writeListLock);
```

Fig. 1. Sample Device Driver Code from [BR01]

code, a ten million line source package will have an expected 10^3 latent defects. To reduce this number, we need to devise complementary analysis and testing techniques. It should also be noted that no-one really knows how many latent defects there *really* are in any given software product. All we can tell is how many of these defects eventually lead to customer complaints, in the years following product delivery. The industry average of 0.5 to 5 defects per one thousands lines of code is based on a count of the typical numbers of those customer complaints. We can suspect that the *true* number of latent defects is at least an order of magnitude higher, more likely in the range of 0.5 to 5 defects per one hundred lines of source code. Looking for latent software defects, then, is not quite like looking for needles in a haystack. Almost any new technique that differs sufficiently from traditional testing should be expected to intercept enough extra defects to justify its application. How could we devise such alternate techniques?

In this paper we will look at three such techniques that have proven to be effective in the software development process, and we will try to show that these techniques have interesting new applications that can be understood in terms of automata theory. The three techniques we will discuss are:

□ Static Analysis,
□ Runtime monitoring, and
□ Logic Model Checking.

2 Static Analysis

Static analysis is the art of making predictions about a program’s runtime behavior based on a direct analysis of its source code. Much of the work in this area is based on the foundational work in abstract interpretation by Cousot and Cousot, e.g. [CC76].

As a motivating example we will take a look at a snippet of C device driver code, reproduced in Figure 1. The code is slightly abbreviated from the original that appeared in [BR01]. We have highlighted three procedure calls: one call on **`lock`** and two on **`unlock`**. As in [BR01] we want to check if this piece of code respects the normal locking discipline that says that it would be an error if a single thread could:

□ Attempt to lock the resource when it already holds that lock,
□ Attempt to unlock the resource when it does not currently hold the lock,
□ Terminate execution with the lock still in effect.

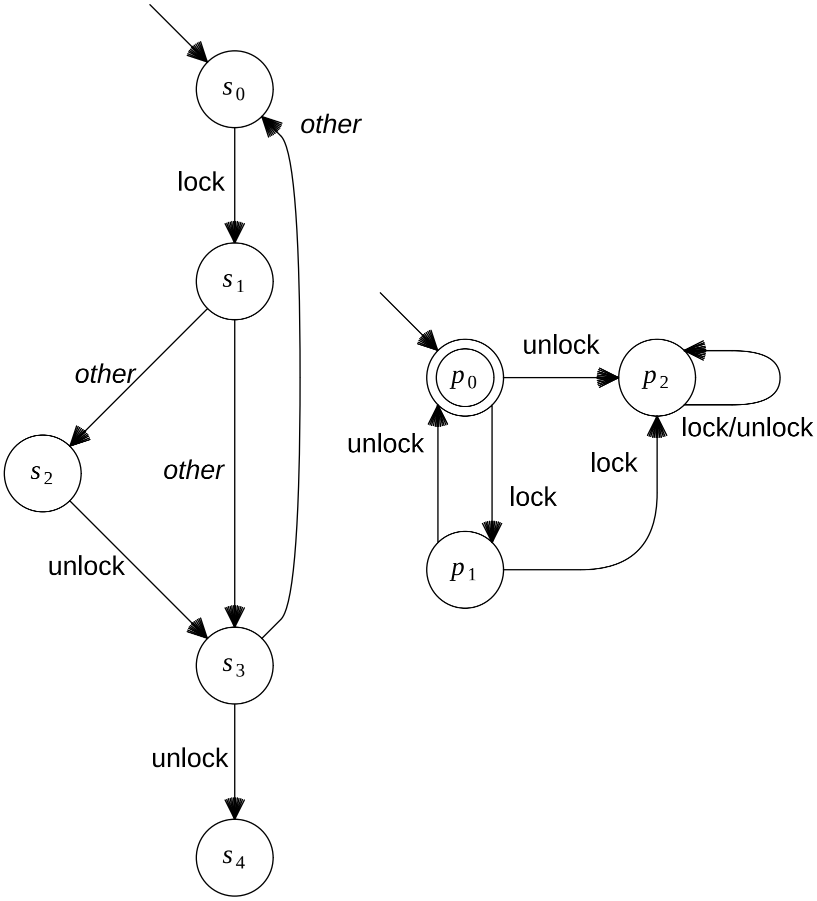

Fig. 2. Control Flow Graph for Figure 1 (left)
and Test Automaton for Locking Property (right)

These types of problems can be found frequently in device driver code, as demonstrated convincingly in [E00].

The first step in the formalization of this code is to generate the control flow graph, as shown in abstract form on the left in Figure 2. The control flow graph can be defined as a labeled transition system: an annotated automaton. Since we are only interested in the occurrences of the calls to lock and unlock, we have marked all other actions as **`other`**. These actions can be considered *skip* statements for the purpose of the initial analysis we will do.

The property we want to check for, the faithful observance of a locking discipline, can also be formalized as an automaton, as illustrated on the right in Figure 2. This property automaton has three states, p_0, p_1, and p_2. The initial state, p_0, is also marked as an accepting state, by which we express that all finite runs that start in p_0 should terminate in p_0. State p_2 is really an error state. Any transition into that state may immediately be flagged as an error.

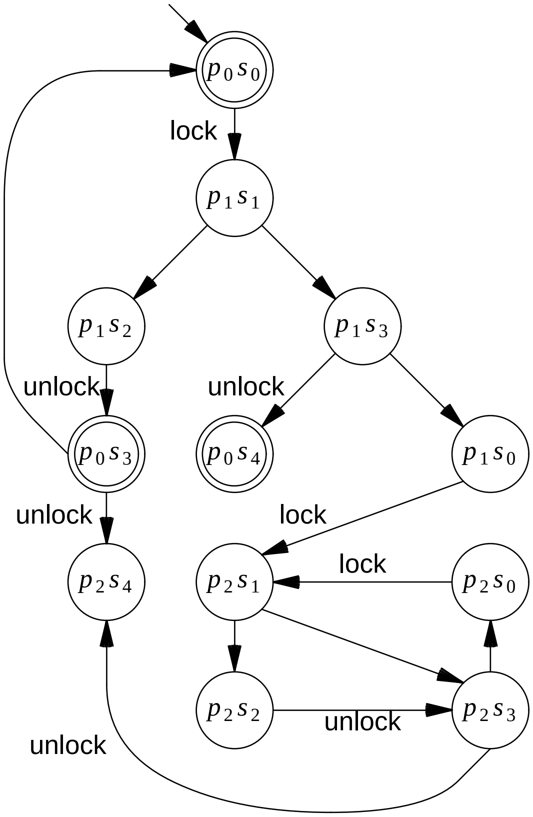

Fig. 3. Product of Test Automaton and CFG Automaton

We will not use this automaton to monitor executions of the program, but to analyze the source code directly. To do so, we in effect try to match the test automaton against the abstracted control flow graph. The formal mechanism for this is to compute the synchronous product of two automata: the automaton that corresponds to an abstracted version of the control flow graph and the automaton for the property.

The product automaton, illustrated in Figure 3, can be defined in a standard way as a synchronization on labels, where **`other`** is treated as a silent τ move that requires no synchronization. It can be computed without ever attempting to really execute the program from Figure 1.

There are two end-states in this product automaton: $p_2 s_4$ and $p_0 s_4$, but only $p_0 s_4$ is accepting. There are also two strongly connected components: C1 = $\{ p_0 s_0, p_1 s_1, p_1 s_2, p_0 s_3 \}$, and C2 = $\{ p_2 s_1, p_2 s_2, p_2 s_3, p_2 s_0 \}$, only one of which contains accepting states (C1). Any feasible finite path that starts in $p_0 s_0$ and that ends in $p_2 s_4$ would correspond to a violation of the locking property. Similarly, any feasible infinite path that reaches $p_2 s_1$ corresponds to a violation.

By computing the product automaton we can narrow down the search for potential violations of the locking discipline to the analysis of just two types of paths:

$[\, p_0 s_0 \rightarrow p_1 s_1 \rightarrow p_1 s_2 \rightarrow p_0 s_3 \,]^+ \rightarrow p_2 s_4$, and
$p_0 s_0 \rightarrow p_1 s_1 \rightarrow p_1 s_3 \rightarrow p_1 s_0 \rightarrow p_2 s_1 \rightarrow \cdots$

where the superfix + indicates a repetition of one or more times of the immediately preceding execution fragment enclosed in square brackets. Mapped onto executions through just the control flow graph from Figure 2, this reduces to:

$[\, s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \,]^+ \rightarrow s_4$, and
$s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_0 \rightarrow s_1 \rightarrow \cdots$

The shortest example of the first path corresponds in source form to:

```
s0: lock( &devExt->writeListLock );
    nPacketsOld = nPackets;
    request = devExt->WriteListHeadVa;
s1: if (request && request->status)    /* true */
    devExt->WriteListHeadVa = request->nxt;
s2: unlock(&devExt->writeListLock);
    /* ... */
    nPackets++;
s3: if (nPackets != nPacketsOld);      /* false */
    unlock(&devExt->writeListLock);
s4: end
```

Similarly, the second path reads in source form:

```
s0: lock( &devExt->writeListLock );
    nPacketsOld = nPackets;
    request = devExt->WriteListHeadVa;
s1: if (request && request->status)    /* false */
s3: if (nPackets != nPacketsOld);      /* true */
s0: lock( &devExt->writeListLock );
     . . .
```

The analysis problem now reduces to determining whether or not these two path fragments are feasible, i.e., whether or not they could occur in a real execution. The feasibility of the two paths is determined by the conditionals that are evaluated along the path. For the first path this means that we should check if the conditionals at s1 and s3 can evaluate to *true* and *false* respectively, given the history of steps that precede them.

In general, this problem is undecidable, so we have no hope of building a general algorithm that can reliably come up with the correct answer in each case. In many practical cases though even partial knowledge of the semantics of C suffices to resolve the question. There is insufficient information to determine if the conditional at s1 could be *true* or *false*, since we do not know the value of the variable `request`, which is derived from `devExt->WriteListHeadVa`. We can, however, tell that the condition at s3 could not possibly evaluate to false in this path, given the only two preceding manipulations of the variables involved. If we reduce the path to just the access to the two variables that appear in the condition this becomes clear:

```
    nPacketsOld = nPackets;
    nPackets++;
s3: if (nPackets != nPacketsOld);    /* false */
```

Clearly, the condition must evaluate to *true*, not *false*, in this context.

For the second path we can come to a similar conclusion. Reducing it to the manipulations of the variables from the second condition we get:

```
    nPacketsOld = nPackets;
s3: if (nPackets != nPacketsOld);    /* true */
```

which again is infeasible in any context. The resolution to these two cases can be done quickly with a theorem prover, or with decision procedures for Pressburger arithmetic, such as the Omega tool from [K96] or the Newton tool described in [BR01]. In the Bell Labs tool UNO, a simple builtin decision procedure is used for the most commonly occurring cases [H02]. Since quick responses from the analyzer are as important as the generation of definitive results for all cases that are in principle decidable, UNO errs on the side of caution and restricts itself to the analysis of only the cases that can be decided most efficiently.

2.1 User-Definable Properties

The UNO tool, defined more fully in [H02], is an attempt to construct a static analyzer that can accept a range of user-defined properties, within the bounds of what is expressible in its property specification language. The basic propositions that can be expressed in this language can refer to data-flow tags that are computed by the tool and attached to the nodes in the control flow graphs of each function. Automata specifications are constructed from these propositions with the normal control-flow syntax of ANSI-C. The locking discipline, for instance as defined by the automaton from Figure 2, is easily expressible by this language, but much richer sets of properties can also be defined. The algorithms used in UNO are relatively straightforward. The tool computes product automata (cf. Figure 3) and performs basic path analysis. It has three basic properties built-in that allow the tool to hunt down what are often said to be the most commonly occurring types of defects in C programs: use of *u*ninitialized variables, *n*il-pointer dereferencing errors, and *o*out-of-bound array indexing errors.

The notion that one can allow the user to freely define program specific properties as automata, and perhaps even derive such automata from logic formulae, to perform *static* analysis is in itself not new. As far as we can tell the first to propose the use of finite state machines to specify properties for static analysis was W.E. Howden [H87]. A version of this idea was used more recently in Engler's work [E00]. In [BR01] the idea of user-defined properties is also used in a clever way that makes it possible to connect static analysis to an approach that is based on logic model checking.

All approaches we discuss here have strengths and weaknesses. If we have to pick a weakness for the approaches based on static analysis it would have to be the decidability problem: determining whether or not paths are feasible. Significant progress has been made on tackling this problem, as noted starting with the work of Cousot and Cousot. The work in this area is already exploited in some very effective commercial tools, such as Polyspace in France, and KLOCwork accelerator in the US.

3 Runtime Monitoring

There are two ways in which we can attempt to side-step the decidability issue that has to be confronted in approaches based on static analysis. One way is to execute the code directly for specific input values, and then monitor the execution for compliance with the requirements. The second way is to execute the code not directly, but symbolically, in abstract form. Admittedly, these approaches too have their weaknesses. For the first approach it is the inability to secure proper coverage, and for the second approach it is computational complexity. We will come back to these issues below. In this section we will discuss direct execution methods based on runtime monitoring, and in the next section we will discuss abstraction methods. Our aim again is to see if these methods can be linked to a standard automata theoretic framework.

The basic idea in runtime monitoring is again simple. We define a test automaton for a property of interest and run it along with an executing system. To be able to check for compliance of the system execution with the test automaton we now instrument the source code to emit signals of all events of interest (i.e., all events that appear in the property to be tested). The monitor can run as a separate process, perhaps even remotely, and tracks the system execution with the help of the signals emitted by the running program. It updates the state of the property automaton, and flags errors as soon as they can be detected.

To do so for the locking property we will have to instrument the code in such a way that not only the basic calls to lock and unlock are recorded, but also the parameters that are passed to those procedures.

Our automata now become extended automata, and can store some data (i.e., references to the locked objects). A clever use of this feature was made in the Eraser algorithm, described in [S97]. The purpose of the algorithm is again to detect locking violations by runtime monitoring, but this time the analysis is focused on the detection of potential data races that are not prevented by the use of locking primitives. The source code is now instrumented not to directly flag calls to lock and unlock, but to flag read and write accesses to shared objects. Each signal to the runtime monitor indicates the object that is referenced, whether the event is a read (r) or a write (w), and the set of locks that is held by the executing process at the moment of access. The automaton used in the algorithm is shown in Figure 4.

The automaton records the transitions that the monitor will make for a single specific shared object. (Each object is monitored by a separate automaton.) We have marked each action in Figure 4 with one or more numbers, with the following meaning.

- □ A zero mark (0) indicates a read (r) or write (w) access by the *first* thread to access the shared data object. A write access in initial state s_0 corresponds to an initialization event. It is an error for a shared object to be read before it is initialized. To reflect this, we added a transition to an error state e.
- □ Transitions marked with a one (1) indicate the first access to the shared object by a new thread (i.e., different from the thread that initialized the object).
- □ On all transitions marked with a two (2) the set of locks held by the current thread is recorded in set *L*.
- □ On all transitions marked with (3) *L* is assigned a new value that is equal to the intersection of the set of locks held by the current thread and the old value of *L*.
- □ The transition marked with (4) is taken when lockset *L* becomes empty, causing

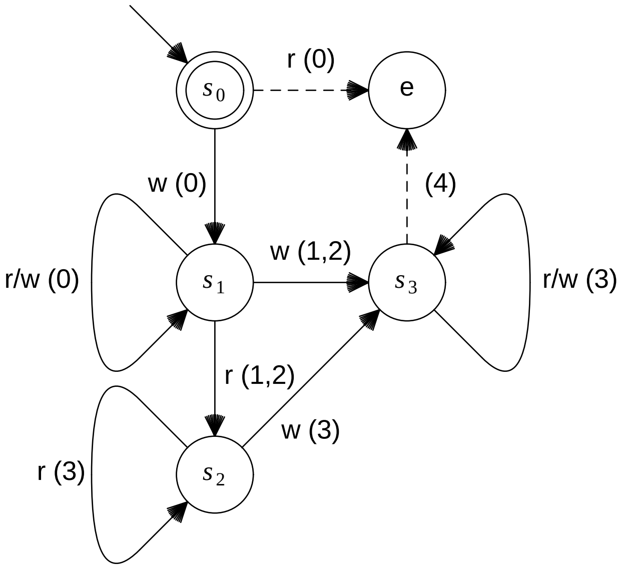

Fig. 4. Eraser Automaton from [S97] (error state e added.)

an error to be reported by the runtime monitor.

Locking violations are only reported in state s_3, which can only be reached if at least two different threads have attempted to write to the shared object. The reason is clear: it is not an error if the shared object is only updated by a single thread, while all competing threads restrict themselves to read access.

The Eraser algorithm has the attractive and curious property that a lock violation need not actually occur for the monitor to be able to detect that it *could* occur in some future execution of the code.

Algorithms based on runtime monitoring have been included in successful commercial tools. The Eraser algorithm, for instance, is included in Compaq's Visual Threads tool [H00]. None of the widely used runtime monitoring tools appear to allow for user-defined properties though, an extension that should readily be possible. We will explore such extensions in a little more detail in the next subsection.

3.1 Liveness

The methods sketched so far use standard finite automata defined over finite strings, or executions. They can capture safety, but not liveness properties. It is possible to extend the capability of a runtime monitor to liveness properties, capturing also more general properties expressed in linear temporal logic.

Consider a system requirement R expressed as an LTL formula. The negation of R, capturing the possible violations of R, can be converted into a Büchi automaton B with a standard procedure, e.g. [GP95]. As before, runtime monitor T will track an execution of the system under test, call it U, with the help of automaton B. This time, though, the transitions in the automaton are labeled not with events from the executions in U, but with boolean expressions on an abstract representation of the system

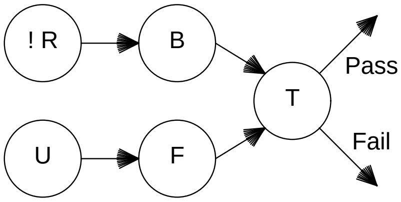

Fig. 5. Runtime Monitoring Context for Checking Liveness Properties
R: LTL requirement; B: test automaton; T: runtime tester;
U: system under test; F: abstraction function.

state of U itself. This context is illustrated in Figure 5.

The abstract system state of U is computed by an abstraction function F. The abstraction function must preserve all information that is relevant to the checking of requirement R, but no more. In many cases, this abstract representation of the system state of U can be considerably smaller than the full representation of the system state.

The source code from system U is now instrumented to invoke function F at specific points in its execution. Each time function F is invoked, it passes its results to the runtime monitor T, which uses it to update the state of automaton B. After each such transition T can declare either a *pass* or a *fail* result, where the *pass* results will typically be silent, and the *fail* results could lead to an abort.

A *pass* result means that T has not yet been able to detect a potential violation of requirement R; a *fail* result means that a potential violation was found. Once a *fail* result has been issued, this result persists throughout the remainder of the execution, until system U is reinitialized (reset or restarted). It is also possible that T loses track of the execution, being unable to construe a counter-example to the requirement. In that case T will move to a state where only *pass* results are issued for the remainder of the execution, again until system U is reinitialized.

The cause of a *fail* result can be the existence of a true implementation error in U, but it can also be an error in the definition of the requirement, an error in the definition of the abstraction function F, or an error in the determination of the invocation points for F. Since a *fail* always produces an execution trace, the source of a false negative can be analyzed with the help of that trace and repaired. As in all tests, an absence of failure does not imply that requirement R cannot be violated by executions of system U. It means that U did not exhibit any violations of R that were detectable under abstraction F.

We can be brief about the definition of F here. It is a critical step, but no different from the problem of defining abstraction functions in more standard applications of logic model checking. The abstraction function will in general be property sensitive, to assure that everything that is directly or indirectly visible to the property is preserved. F can be derived with the help of static analysis techniques, but it can also be determined experimentally by a human tester.

A more critical point in the definition of the runtime monitor T is to determine how it could reasonably conclude Büchi acceptance based on only finite executions of U. To do so T must be able to detect the occurrence of execution cycles through Büchi acceptance states in automaton B. We will distinguish two cases. The simple case

```
Verdict
function d_check (Ustate u)     # B is deterministic
{
        if ( t == fail ) { return FAIL }
        if ( t == pass ) { return PASS }

        if Accepting(t)
        {
                if state u is in set States(t)
                {
                        t = fail
                        return FAIL
                } else
                {
                        States(t) += u
        }       }

        if empty Succ(u,t)
        {
                t = pass        # B is incomplete
        } else
        {
                t = Succ(u,t)
        }

        return PASS
}
```

Fig. 6. Checking Liveness with a Deterministic Test Automaton

where B is assumed to be deterministic, and the harder case where B can be non-deterministic. In neither case do we require B to be completely defined.

For deterministic B, a simple algorithm for T is shown in Figure 6. The notation used is as follows:

```
t:             the current state of automaton B
u:             last observed abstract state of system U
Succ(u,t):     function returning the successor state
               in B, if any, given current state t and
               abstract system state u
States(t):     a set of abstract system states associated
               with t
Accepting(t):  boolean function that returns true if t is
               a Büchi acceptance state
x += y:        add element (or set) y to set x
```

Assume that F invokes the function *d_check* with parameter *u* encoding the abstract representation of the current system state of U. Function *d_check* returns a *pass/fail* verdict for each such call. The tester T maintains the automaton state for B in a global variable *t* whose value persists across calls. Variable *t* can have two special values. The special value *fail* means that a failure has been detected and the automaton will now issue only *fail* results until T is reinitialized. The special value *pass* means that the automaton was incomplete, causing T to lose track of the execution. Only *pass*

```
Verdict
function nd_check (Ustate u)    # B non-deterministic
{
        if ( t == fail )        { return FAIL }
        if ( t == pass )        { return PASS }

        xt = empty              # next state set in B
        for each ot in t        # current states in B
        {
                if {u,ot} in Pairs(ot)
                {
                        t = fail
                        return FAIL
                }

                if Accepting(ot)
                {
                        Pairs(ot) += {u,ot}
                }

                for each nt in Succ(u,ot)
                {
                        Pairs(nt) += Pairs(ot)
                        xt += nt
                }
        }

        if empty xt
        {
                t = pass        # B is incomplete
        } else
        {
                t = xt
        }

        return PASS
}
```

Fig. 7. Checking Liveness with a Non-Deterministic Test Automaton

results will be issued now, again until T is reinitialized.

The essence of the algorithm is that for every visit to a Büchi acceptance state in B the tester remembers the encoding of abstract system state u. A repeat visit to the state with the same encoding u points at a potential cyclic execution of U that violates the requirement (by satisfying its negation in B).

The same type of algorithm could be used for a non-deterministic automaton B, but it may lead to more false negatives than necessary. The algorithm in Figure 7, due to Mihalis Yannakakis, is more precise. The following notation differs from the one used in Figure 6:

```
t:         the set of current states of automaton B
Pairs(t):  a set of pairs {u,b} associated with automaton
           state t of previously seen combinations of
           abstract system state u and accepting automaton
           state b
```

Pairs of abstract system states and accepting automaton states that have been encountered in the current execution are now propagated forward in the execution, until a repeat is seen. The algorithms in Figures 6 and 7 are claimed to have the following properties.

- If system U cannot violate requirement R, then in any execution of U the tester T will report *pass* at every step.
- If any monitored execution of system U violates requirement R, then the tester will report *fail* at the earliest point in the execution where the violation can be conclusively detected under abstraction F.
- If system U can violate requirement R, but the monitored executions of U do not demonstrate this, then tester T may or may not report a *fail*.

4 Logic Model Checking

The best known example of an automata based approach to software verification is logic model checking. In applications of model checking to software verification, there are some flies in the ointment again that often prevent us from claiming more than best effort results. For fundamental reasons, model checking techniques cannot be applied blindly to arbitrary program code written in the currently popular programming languages (cf. [S65]). To apply a model checking algorithm we need to create an abstract, finitary, representation of the source code: a verification model. The verification model is traditionally constructed by hand, and is therefore subject to human error. Human error can introduce both defects that the real system does not have and it can hide defects that the real system does have. Errors of the first type are mostly harmless, since they will trigger false negatives that can be repaired after an analysis of an error trace. Errors of the second type, however, can lead to false positives, which are much harder to diagnose.

Another, often forgotten, potential source of error is in the definition of the properties. Defining properties is unavoidably a human effort, and it is therefore quite predictable that they will in some cases be incorrect. Again, it is easily repaired if a bad property leads to a false negative, but we should also consider the possibility of false positives.

To reduce one source of potential error we can try to extract automata models mechanically from source code. There have been several promising attempts to do so. For Java, this includes the Bandera toolset [CD00] from Kansas State University, and the Pathfinder tools from NASA Ames Research Center [HP00,VP00,VH00]. For C this includes the Bebop toolset from Microsoft [BR01], and our own FeaVer tool, e.g. [HS00], [HS02]. The judicious application of abstraction techniques is key to the success of these techniques. One can use property based slicing [T95,HD00], predicate abstraction techniques [VP00], static analysis, and theorem proving techniques to justify the abstractions.

In the application of FeaVer, abstraction functions are recorded in a lookup table that acts as a filter for the source code. Abstraction is only applied to basic statements and conditionals; the control-flow structure of the source code is preserved. To apply the

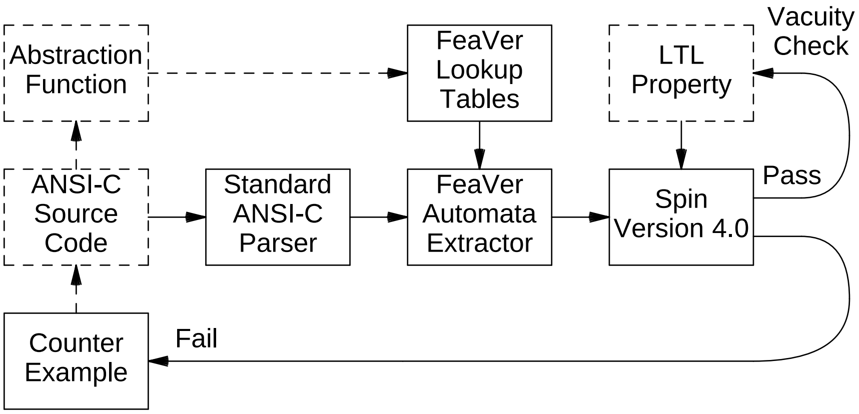

Fig. 8. FeaVer Model Extraction Framework
(Dashed arrows are user-defined steps; solid arrows are automated steps)

abstraction and generate the system model, the source code is first parsed, with a standard compiler front-end. FeaVer uses a simple ANSI-C parser to perform this step. The parse tree for each function in the source code is converted into a control-flow graph, which becomes the basis for model extraction, as illustrated in Figure 8. The edges in the control-flow graph correspond to the basic statements in the source language (e.g., assignments and function calls). Most nodes will have a single outgoing edge, encoding normal sequential control flow. Nodes without any outgoing edges correspond to function termination points. Nodes with more than one outgoing edge correspond to decision points, and are labeled with the criterion that is to be used for the selection of an outgoing edge. For a standard if-then-else construct, the condition is an expression, and there are two outgoing edges: one labeled *true* and the other labeled *false*. For C-style switch statements, there can be more outgoing edges, each labeled with a different numeric value for the switch expression.

The abstraction function in FeaVer is applied only to the labels on the edges in the control-flow graph. The abstraction can result in certain operations to be hidden completely (as if they are sliced away from the program text), others to be modified in accordance with the coarsening of selected data types, and still others may be preserved as is. Conditions can similarly be removed completely, and replaced with non-deterministic selections, they may be modified in accordance with data type mappings, or they can be preserved as is.

4.1 Functions to Automata

The FeaVer model extraction process targets C functions as the primary focus of abstractions. As part of a FeaVer test harness setup (not shown in Figure 8), the user must define the specific C functions that are to be converted into SPIN automata models, and provide a context for those models. Typically there is one main control function per thread of execution, and that control function will be targeted for model extraction. The functions converted into SPIN models will often call subsidiary functions to perform smaller computational chores. Those subsidiary functions are best left in the code, and invoked as-is as embedded C-code, or abstracted with the help of a stub-function. The notion used here is that the definition of functions is the main vehicle for a programmer to define abstractions in C programs. By linking the model extraction mechanism to this feature, we can adapt and reflect the programmer chosen abstractions into the SPIN verification models.

The definition of an abstraction function, or of the FeaVer lookup tables, is one of three user-defined items in the verification process that remains subject to human error (cf. Figure 8). The definition of an abstraction function is one step removed from the code itself, and, in the applications we have performed so far has proven to be significantly simpler and less time consuming than the manual definition of the entire automaton model. The opportunity for error in this step should therefore be reduced, although it is hard to quantify this reduction precisely. Perhaps more important than the above is the gain in efficiency that is achieved through the model extraction process. As indicated in Figure 8, once an abstraction is defined, the remainder of the model checking process can be automated completely. This automation has the very noticeable benefit that the verification of evolving source code becomes virtually push-button, and can be repeated immediately after every source code revision by the programmers.

4.2 Vacuity Checking

When the model checker returns a *pass* verdict, it means that it was unable to construct a counter-example to a correctness claim. Since the claim is derived from a user-defined property this still leaves the possibility that the claim is ill-defined and vacuously true. In the FeaVer application to the verification of the PathStar® call processing code [HS00], we supported a simple vacuity check by generating a graphical version of the ω test automaton, coloring each state in the automaton remained unreached in all model checking attempts for the property from which it was generated. We normally expect that all but a few final states in the automaton are reached. If only the initial state is reached, or a minority of all the remaining states, this most likely indicates a vacuous result. A more thorough method of vacuity checking can be found in [KV99].

4.3 Main Benefits

The FeaVer tool grew out of an attempt to come up with a thorough method to check the call processing software for a commercial switching product, called the PathStar® Access Server. We pursued this approach over a period of eighteen months, working with the developers of the target software from the initial design for the code in 1998 to after commercial sales for the product began in 2000. The project was successful in demonstrating that the approach based on model checking could intercept a significantly larger fraction of the design and coding errors in the software than traditional testing techniques: we measured a difference of approximately one order of magnitude in effectiveness, as measured by the number of defects found with each method [HS00]. We also succeeded in realizing our original goal of automating almost the entire verification trajectory by driving all our software from a web-browser that gave access to a database of verification results, and allowed users to enter new properties, start verification runs, and retrieve the results of such runs. We have also applied FeaVer in several smaller projects with similar results, e.g. [GH02].

Although the FeaVer approach side-steps the need for the time-consuming construction of verification models, we have found that some of the remaining tasks can still be challenging. Once a FeaVer test harness has been defined, the remainder of a verification exercise is relatively simple and requires no substantial investments of time or skill, even if the source code continues to evolve. But the initial construction of a test harness requires thought, and experimentation [HS02]. The test harness defines which functions are converted into SPIN process threads, how these threads are connected (i.e., how internal communication from the C code is lifted into a communication mechanism that is visible to SPIN). Most important in the definition of the test harness, though, is the definition of abstraction functions that populate the FeaVer lookup tables, and that determine which state information from the C code should be tracked in the SPIN models. If we could succeed in simplifying, or even automating, these tasks, the value of this approach would be increased significantly.

5 Conclusion

The aim of this paper is to show that many of the new approaches to software testing are based on, or can be understood as applications of, automata theory. User-defined temporal properties may be checked with approaches based on run-time monitoring, static analysis, and of course in standard applications of logic model checking. In logic model checking considerable gains can be made by finding ways to extract automata models directly from program source code. Not all problems are solved in this domain, but perhaps we are beginning to see the types of checking capabilities that might be ahead.

Acknowledgements

The author is grateful to Mihalis Yannakakis and Klaus Havelund for many inspiring discussions on the topics covered in this paper. The work on FeaVer was done jointly with Margaret Smith and greatly benefited from the collaboration with Ken Thompson and Phil Winterbottom in the first application to PathStar®.

References

[BR01] T. Ball, S.K. Rajamani, Automatically Validating Temporal Safety Properties of Interfaces, *Proc. SPIN 2001 Workshop on Model Checking of Software*, Springer LNCS 2057, May 2001, Toronto, pp. 103-122.

[CD00] J. Corbett, M. Dwyer, et. al. Bandera: Extracting Finite-state Models from Java Source Code. *Proc. ICSE 2000*, Limerick, Ireland.

[CC76] P. Cousot, R. Cousot, Static Determination of Dynamic Properties of Programs, In B. Robinet, (Ed.), *Proc. 2nd Int. Symp. on Programming*, Paris, France, April 1976, pp. 106-130.

[H00] J.J. Harrow, Runtime checking of multithreaded applications with Visual Threads. *Proc. SPIN 2000 Workshop on Spin Model Checking and Software Verification*, Springer LNCS 1885, August/Sept. 2001, Stanford University, pp. 331-343.

[HD00] J. Hatcliff, M.B. Dwyer, and H. Zheng, Slicing software for model construction, *Journal of Higher-Order and Symbolic Computation*.

[E00] D. Engler, B. Chelf, A. Chou, and S. Hallem, Checking system rules using system-specific, programmer-written compiler extensions. *Proc. 4th Symp. on Operating Systems Design and Implementation (OSDI), Usenix Organization, San Diego, CA., Oct. 22-25, 2000.*

[GP95] R. Gerth, D. Peled, M. Vardi, P. Wolper, Simple On-the-fly Automatic Verification of Linear Temporal Logic, *Proc. Symp. on Protocol Specification Testing and Verification*, Warsaw, Poland, 1995, pp. 3-18.

[GH02] P.R. Gluck, G.J. Holzmann Using Spin Model Checking for Flight Software Verification, *Proc. 2002 Aerospace Conference*, IEEE, March 2002, Big Sky, MT, USA.

[HP00] K. Havelund, T. Pressburger Model Checking Java Programs Using Java PathFinder *Int. Journal on Software Tools for Technology Transfer*.

[H97] G.J. Holzmann, The model checker . *IEEE Trans. on Software Engineering*, Vol 23, No. 5, pp. 279-295, May 1997.

[H01] G.J. Holzmann, Economics of Software Verification, *Proc. Workshop on Program Analysis for Software Tools and Engineering*, ACM, Snowbird, Utah, USA, June 2001.

[HS00] G.J. Holzmann, and M.H. Smith, Automating software feature verification, *Bell Labs Technical Journal*, April-June 2000, pp. 72-87.

[HS02] G.J. Holzmann, and M.H. Smith, FeaVer 1.0 User Guide, Technical Report, Bell Labs, February 28, 2002, 64 pgs.

[H02] G.J. Holzmann, Static source code checking for user-defined properties, *Proc. IDPT 2002*, 6th World Conference on Integrated Design & Process Technology, Pasadena, CA, USA, June 2002.

[H87] W.E. Howden, *Functional Program Testing and Analysis*, McGraw Hill, 1987.

[K96] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, *The Omega calculator and library*, Version 1.1.0. Technical Report November 18, 1996, University of Maryland.

[KR88] B.W. Kernighan, and D.M. Ritchie, *The C Programming Language, 2nd Edition*, Prentice Hall, Englewood Cliffs, N.J., 1988.

[KV99] O. Kupferman, M.Y. Vardi, Vacuity detection in temporal model checking, Conf. on Correct Hardware Design and Verification Methods, Springer-Verlag, LNCS 1703, 1999, pp. 82-96.

[AM02] L. Amster, D.L. McClain (Eds.), *Kill Duck Before Serving, Red Faces at The New York Times*, Publ. St. Martin’s Griffin, New York, 2002, 172 pgs.

[S97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, Vol. 15, No. 4, pp. 391-411, 1997.

[S65] C. Strachey, An impossible program, Computer Journal, Vol. 7, No. 4, January 1965, p. 313.

[T95] F. Tip, A survey of program slicing techniques. *Journal of Programming Languages*, Vol. 3, No. 3, Sept. 1995, pp. 121-189.

[VP00] W. Visser, S. Park, and J. Penix, Applying predicate abstraction to model checking object-oriented programs. *Proc. 3rd ACM SOGSOFT Workshop on Formal Methods in Software Practice*, August 2000.

[VH00] W. Visser, K. Havelund, G. Brat, and S. Park, Model checking programs. *Proc. Int. Conf. on Automated Software Engineering,* Sept. 2000.