From Code to Models

Gerard J. Holzmann
Bell Laboratories, Lucent Technologies
600 Mountain Avenue, 2C-521
Murray Hill, New Jersey, USA
gerard@research.bell-labs.com

Abstract

One of the corner stones of formal methods is the notion that abstraction enables analysis. By the construction of an abstract model we can trade implementation detail for analytical power. The intent of a model is to preserve selected characteristics of real-world artifact, while suppressing others. Unfortunately, practitioners are less likely to use a modeling tool if it cannot handle real-world artifacts in their native format. The requirement to build a model to enable analysis is often seen as a verdict to design a system twice: once in a verification language and once in an implementation language. Because the implementation phase cannot be skipped, verification is often sacrificed.

In this paper we will consider a way to avoid this problem by automating the extraction of verification models from implementation level code. The user now provides only model extraction rules, or abstractions, rather than full-scale models.

1. Introduction

Models are used in most engineering disciplines. They come in different forms. A model can be a mathematical theory, a physical entity, or a mere guiding mental image in the mind of a designer. The purpose of a model is to facilitate analysis, either explicitly or implicitly.

Models are *abstractions* of real world artifacts. A designer only takes the trouble to build a model when it is easier, cheaper, or faster to analyze the model than it is to analyze the real world artifact itself. A model that is more complex than the artifact that it describes would be comparable to the summary of a book that is longer than the book.

The effective use of abstraction is the key to the successful construction of models. The construction of an abstract model, in turn, is the key to successful analysis.

A model should not only be simpler than the artifact being modeled, it should also be simpler to *construct* than that artifact. If the construction of a model is perceived to be too complex, it will be all too tempting to forgo the construction of the model, and make do with more basic types of analysis.

In this paper our focus will be on the construction of analytical models for software artifacts (i.e., program source code), in particular software artifacts for *concurrent* systems. In line with the above observations, we can state two conditions that should be fulfilled for this work to be successful. *First*, the model must enable a type of analysis that cannot more easily be obtained by other means. *Second*, the construction and analysis of the model must be (at least perceived to be) a relatively minor task compared with the construction of the actual system. The first condition is easy to fulfill; the second, however, is much harder to satisfy. We will consider what it entails.

2. Model Construction

Logic model checking tools can be quite powerful when used to analyze software artifacts. Especially in the area of distributed software systems design, a model checker can generously outperform conventional approaches to software testing in the detection of concurrency related defects [4,5]. A drawback of the use of model checkers, however, is that the construction of the verification model for any non-trivial system is often a non-trivial task, even for experts.

To construct a model, a verification expert typically consults with the designers of the system being studied, reads extensive systems documentation, and in some cases studies the source code of the application to develop an understanding of the intended design. This process minimally takes days, often weeks, and sometimes months. The software itself, meanwhile, continues to evolve. To isolate the model as much as possible from the continuing changes, a verification expert has to make a choice. One can either choose a level of abstraction that shields the model from lower level change, using *ad hoc*, principles for determining a satisfactory level of abstraction, that themselves often have to change to keep pace with the changing software. Or one can choose to maintain a more precise model based on unchanging principles of abstraction, at the prize of constantly having to be on the alert for changes in the implementation that

might affect the model. Many will choose the first option, but this choice too has unfortunate consequences.

One consequence is that the abstraction rules are often ill defined, poorly documented, and frequently changed. If the rules are not changed to keep track of the evolving systems design, the verification effort runs the risk of outliving its usefulness: generating results for a design that no longer exists. An application like this is therefore only of value when applied to highly critical software components where design work can be frozen sufficiently long for a formal analysis to be completed, and with sufficient visibility to entice verification experts to take part in the effort. Most industrial software projects, however, are not of this type and a different approach is called for.

In what follows we will describe the essence of the approach taken in the Bell Labs' FeaVer project. The aim of this project is to explore the possibilities for automated model extraction from C programs.

There are a number of related projects that have very similar goals. In the Slam project at Microsoft, for instance, Tom Ball and his colleagues are building the BeBop model checker for analyzing Boolean abstractions of C programs [1]. In the Bandera project at Kansas State University, Matt Dwyer and John Hatcliff target Java programs [2], as do NASA/Ames researchers Klaus Havelund and Willem Visser in their Pathfinder project [3,9]. The philosophy of automated model construction to provide model checking capabilities to programmers is similar in all these efforts.

3. Abstraction: Control and Data

Our objective is to allow the designer of a software artifact to specify a set *rules* that can be used to guide the mechanical extraction of verification models from implementation level code. By specifying rules rather then the result of applying those rules, we can achieve two things:

1. the rules are explicitly documented, and
2. the verification effort can more easily track an evolving code base.

The rules we use should themselves be able to fulfill two closely related roles: defining both abstractions and syntactic conversions from program-code to verification models (or *mappings*). In what follows we will mostly talk about developing a framework for the specification and use of *general conversion rules*. Abstractions can be expressed naturally within this framework so we will not treat them separately here.

The conversion rules that we will use for model extraction may need maintenance when the target code evolves. Our intent, and initial experience is, that it is simpler to maintain a concise set of conversion rules than it is to maintain a detailed, manually derived verification model. The rules, in effect, define a *generator* for the verification model, without detailing the structural aspects of either the verification model or the source code.

To be specific, we limit the following discussion to programs that are written in the C programming language. A software program, then, consists of the definitions of a *control structure* and a set of *data objects*. The data in a program definition can consist of either statically or dynamically allocated data objects, as well as implicitly allocated data, such as the data that exists on the program call-stack.

To apply finite state model checking techniques, we must make sure that the model that is generated from a program is finite, even when the original program is not. Fortunately, in one case this will be relatively simple: in C programs the definition of the control structure is always finite. (Even a recursive program is defined by a finite control structure.) This much is good news. A second piece of good news that we will try to exploit is that the complexity of a C program is contributed almost exclusively by implicitly and explicitly declared data objects, not by the control structure.

It would therefore be beneficial if we could focus our model extraction rules on the use of data, rather than program control structures. This means that we have to be able to separate control and data. The control structure from a program can in most cases be preserved in the extracted models, with only minor exceptions (e.g., to eliminate the possibility of unbounded recursion, which can be handled through conversion rules).

4. Defining a Verification Context

Software testing always takes place in a specific context, with sometimes unstated, and frequently changing, assumptions about the conditions for success or failure. The assumptions that are often implicitly made concern the likelihood of specific types of failures, the possible inputs to a system, and the range of possible responses from connected processes and devices. To setup reliable and reproducible test-cases for a distributed system's test, these choices must be made explicit, documented, and changes tracked. We do so in a verification context, or as we shall call it a *verification test harness*.

The test harness has to be specified in some format. For a C application, a natural choice would be to use the language C itself. This choice would not be incompatible with the framework that we are describing, since clearly once we can mechanically extract Spin verification models [5] from the C source code of an application, we can also do so for other fragments of C. In many cases, though, it is more convenient to express the verification

context in the language of the model checker itself. One benefit of this is that we can exploit the model checker's built-in support for non-determinism to generalize (weaken) our assumptions about the verification context. A potential drawback is that the builder of the verification test harness has to learn a new language to fully utilize this potential.

The verification test harness has to be built just once for each application. If built correctly it will be largely immune to changes in the source code of the application, as it evolves through its design cycle, and through normal system maintenance.

A small example can illustrate what the main components of a *verification test harness* are. We will consider the following implementation in C of a simple send-response handshake between two processes. The code might look as follows:

```
extern const int p0;
enum msg_type { Msg, Ack, TimeOut };

void handshake(void)
{    int resp;

     send(p0, Msg);
     set_timer(16000); /* msec */

     resp = wait_recv();
     switch (resp) {
     case Ack:
            reset_timer();
            . . .
            break;
     case TimeOut:
            . . .
            break;
     default:
            reset_timer();
            error("bad input");
            break;
     }
}
```

The executing process first sends a message Msg to a peer, which is identified here by `p0`. The process then sets a timer to, say, twice the average round-trip transmission delay, and waits for a response. A timer process, assumed to have been defined elsewhere, starts counting down as soon as the `set_timer` function is executed. If the response arrives in time, the timer is reset. If the count in the timer reaches zero before the response arrives, the timer generates a `TimeOut` message that is now processed by the handshake process.

First we can observe that the control structure is indeed finite, contributing just a handful of control states, which for the time being we can think of as the possible values of the program counter within this procedure.

There are various types of data used. The values of `p0` and `Msg` are declared as constants, and the return value of the wait_recv() function is declared as a local integer. Another data object, declared externally to this procedure, must be used store the value of the implied count-down timer. Clearly, the count-down timer can contribute up to 16,001 possible states to the verification model that we might try to extract from this program fragment (the value 0 also counts). The variable that records the return value of `wait_recv` could contribute even more (we can only tell how many by inspecting that procedure).

The verification context for this piece of code clearly includes assumptions about the behavior of the peer process, the timer process, and the here unspecified implementation of the wait_recv() function. Some of this information may be available to use, but even if it is it can be wise not to rely on it too strictly. After all, this information can be subject to change, and it would be better if the correctness of this piece of code did not depend in too great detail on the particulars of other system components.

In the definition of the verification test harness we formalize conservative assumptions about the behavior of the unspecified components in the form of abstract "test-drivers."

First, let us consider how we can formalize some explicit assumptions about the behavior of the timer process, in Promela [5]. We will assume here that the `Set` and `Reset` messages can originate from only one source: the handshake process. After the `Set` message is received, the timer should enter its count-down cycle, from which it can only exit when either a `Reset` message is received or when the counter reaches zero. When the counter reaches zero, a `TimeOut` message is sent. We will also need to declare a channel structure for the communication between the handshake process and the timer within our test system. The following test driver model records these decisions.

```
chan timer = [0] of { mtype, chan, int };

mtype = {    Msg, Ack, Other,
             Set, Reset, TimeOut };

active proctype timer_p()
{    chan who;
     int cnt;

     do
     :: timer?Set(who,cnt) ->
            do
            :: timer?Reset(who,cnt) ->
                   break
            :: empty(timer) ->
                   if
                   :: cnt > 0 -> cnt–
                   :: else ->
                          who!TimeOut;
                          break
                   fi
     od     od
}
```

Next, we add some conservative assumptions about the possible behavior of the peer process that is to generate

responses to the handshake process. As far as we can tell from the code of the handshake procedure, the peer process either responds to an incoming Msg with an Ack message, or with a random value within the range of the integers. We also need to introduce as part of the infrastructure for the test system two channels for the communication between the handshake process and the peer process. The channel to the peer process will send only symbolic message types; the reverse channel should be able to transmit arbitrary integers. We can record all these assumptions as follows in a test driver:

```
chan p0 = [0] of { mtype };
chan q0 = [0] of { int };

active proctype peer()
{    int n;

     do
     :: p0?Msg ->
            if
            :: q0!Ack
            :: n = 0;
                   do
                   :: n++
                   :: break
                   od;
                   q0!n
            fi
     od
}
```

The integer variable n is used here to generate a random value. The increment of this variable will wrap around its maximum eventually, and cycle through its range. The break from the increment loop can happen non-deterministically at any point in the range, giving the desired generality.

Before we move our attention to the construction of a model for the handshake procedure itself (the focus of the effort), let us consider the test driver code in a little more detail, to see if we can use abstraction techniques to simplify them without loss of generality.

First we can note in the C code for the handshake procedure that, even though the function `wait_recv` may return arbitrarily integer value, only *three* types of values can affect its execution. This means that we can define an abstraction on data object `n` in the model of the peer process from an integer to a constant. Any value other than the values represented by `Ack` and `TimeOut` will do. We introduce a symbolic constant Other for this purpose. This reduces our assumptions for the behavior of the peer process to the simpler:

```
chan p0 = [0] of { mtype };
chan q0 = [0] of { mtype };

active proctype peer()
{
     do
     :: p0?Msg ->
            if
            :: q0!Ack
            :: q0!Other
            fi
     od
}
```

Next let us reconsider the assumptions we made about the timer process. Even though the counter variable used there can have 16,001 possible values, in many cases only *two* sets of values will need to be taken into consideration: zero and nonzero. If we include this assumption, our test driver for the timer process can be reduced as follows. The counter variable is changed from an integer into a Boolean. Once the counter is set the timer may expire and transmit the `TimeOut` message, but it need not do so. When expired the timer cannot send any messages.

```
active proctype timer_p()
{    chan who;
     bool status;

     do
     :: timer?Set(who,_) ->
            status = true
     :: timer?Reset(who,_) ->
            status = false
     :: status == true ->
            if
            :: who!TimeOut
            :: skip /* do nothing */
            fi
     od
}
```

We can simplify this still further by first noting that under Promela semantics the option sequence for a true value of the `status` variable need not be executed at all, and so the `skip` alternative is redundant. Secondly, the true value of the `status` variable can be derived from a non-zero value of the `who` variable. This leads to the following simpler version for the timer test driver:

```
active proctype timer_p()
{    chan who = 0;

     do
     :: timer?Set(who,_)
     :: timer?Reset(who,_)
     :: who != 0 -> who!TimeOut
     od
}
```

Is it worth the effort to go from the first version of the test drivers to the more abstract versions? Most assuredly so. The first version would for all practical purposes render the entire verification effort intractable. Note, for instance, that the 2^{32} states of the integer variable n in the model of the peer process would multiply the state space size by roughly four billion. The integer count-down timer would contributes another multiplication factor of at least five orders of magnitude. The final version contributes just 2 states for the peer process, and 2 states for the timer: a very considerable reduction of complexity. And

we haven't even looked at the handshake procedure itself yet.

Now that we have defined a complete verification test harness, we can direct our attention to the real focus of the verification effort: the construction of a verification model for the handshake procedure. The test drivers were setup in such a way that they encapsulate only conservative assumptions about the context in which we want to verify the behavior of any process executing the handshake procedure. The handshake procedure itself, however, must be rendered more explicitly and more faithfully. It is this procedure only that we would expect to change, during normal system evolution. If the test harness description is right, it will only rarely have to be adjusted. So unlike the components used in the verification test harness, it will be attractive if we can generate the model for the handshake procedure mechanically from the source code.

We may have to perform the model extraction repeatedly, over a long period of time, while the code for this procedure continues to evolve. In this case, tracking changes in the handshake procedure would be simple, because of its modest size. In general, though, the target software may be many thousands of lines of code, not trivially understood in its functionality, and subject to frequent change. In these cases a reliable way of constructing the model and tracking the changes is essential.

5. Model Extraction

The FeaVer model extractor [4] generates the following Promela model from the C source text of the handshake procedure, without any further user input:

```
hidden int TimeOut = 1;
hidden int Ack = 2;
hidden int Msg = 3;
int p0;

active proctype handshake()
{   int resp;

    c_code { send(now.p0,Msg); };
    c_code { set_timer(16000); };
    c_code { Phandshake->resp=wait_recv(); };

    do
    :: c_expr{ Ack == Phandshake->resp };
        c_code { reset_timer(); };
        break; goto C_0
    :: c_expr { TimeOut == Phandshake->resp };
C_0:    break; goto C_1
    :: else ->
C_1:    c_code { reset_timer(); };
        c_code { error("bad response"); };
        break; goto C_2
    od;
C_2: skip;
}
```

The output is generated in the syntax accepted by Spin Version 4.0, which supports a few new types of statements that were inspired by the FeaVer project. The two most important of these statements are `c_code` and `c_expr`, which also appear in the fragment shown above.

A `c_code` statement can encapsulate any fragment of code written in ANSI-standard C. The code fragment is treated as an externally defined state transformer by the model checker. The execution of a `c_code` statement is defined to be atomic and unconditional, just like a Promela assignment, a `printf`, or a `skip` statement, but other than a `d_step` (for which the executability is conditional on the executability of its guard).

The second new primitive, `c_expr`, can encapsulate an arbitrary expression written in the syntax of ANSI-standard C. The evaluation of the expression is required to be free of side-effects on variable values. A `c_expr` statement is *executable* (in the sense of Promela semantics) if and only if an evaluation of the expression returns a non-zero value, otherwise the statement *blocks*.

If the syntax looks forbidding, it is probably good to keep in mind that the `c_code` and `c_expr` statements are not intended to be written by humans, only by model extraction tools such as FeaVer.

A few things are worth noting about the FeaVer generated model. First, the model does indeed reproduce the control structure of the original faithfully, in Promela syntax. It further encapsulates all basic statements and expressions in, respectively, `c_code` and `c_expr` statements, depending on context. To accomplish this, the FeaVer tool makes use of a specially instrumented compiler front-end to parse the C code and to determine its structure.

A second observation is that no abstractions are defined or applied by default here. All abstractions will have to come from user-supplied conversion and abstraction rules that we will consider shortly. Thirdly, and finally, the model is defined at the level of a single procedure, it does not automatically descend into the procedure call hierarchy. Any function or procedure calls that appear inside our target procedure are by default preserved in a `c_code` or `c_expr` wrapper. This gives us the freedom to define special-purpose implementations of these procedures for testing purposes, or to preserve the calls as they are written. It also allows us, to support bounded recursion within the verification models, in native C code, provided that the procedure call stack contains no hidden state information that the model checker would have to be aware of.

All data references that appear inside `c_code` and `c_expr` fragments are of course also stated in C, rather than in Promela. The reference to the local variable `resp` in the FeaVer generated model, for instance, is generated

with a structure prefix that locates the variable in the state vector. The model extractor automatically modifies data references for all local and global data objects encountered in this way.

To fit the generated model into the verification test harness we should now define some explicit conversion rules. We will record these rules in a simple table, with source code fragments to be matched and target conversions. A conversion table can record not just syntactic conversions but also a broad range of general abstraction rules. An abstraction generator could, for instance, be used to generate carefully justified entries for such a table.

Here we will restrict our attention to the minimal syntactic conversions that are be needed to obtain a verifiable system description.

The conversion rules used by the model extractor always apply only to the basic statements and expressions that appear in the C source code, not to control flow constructs. There are just a few such basic statements and expressions in the C source from our example: we can count three separate function calls, one assignment statement, and two stand-alone expressions that are implicitly used in the switch statement. They are:

```
set_timer(16000)
reset_timer()
send(p0,Msg)
resp = wait_recv()
error("bad response")
Ack == Phandshake->resp
TimeOut == Phandshake->resp
```

We can now define optional conversion rules for any subset of these statements, overriding the built-in defaults. The precise interpretation of the interactions with the timer and the peer processes, for instance, are modeling choices that would be difficult to derive purely mechanically. We record these choices it as follows, with four conversion rules:

```
set_timer(16000)     timer!Set(q0,16000)
reset_timer()        timer!Reset(q0,0)
send(p0,Msg)         p0!Msg
resp=wait_recv()     q0?resp
```

We have specified these rules in a tabular format. The table has two columns, the left-hand side giving a source text statement and the right hand side the replacement text that we would like to use instead of the default mappings.

A similar choice can be made to interpret the call on the error routine in the original source as an execution error that we would like to catch. We can do so by defining the following conversion rule.

```
error(. . .  assert(false)
```

We have used a more general pattern here to specify the source code statement to be matched. The usage of the three dots implies that the rule matches all source statements that begin with the prefix "`error(`".

The two conditional expressions, testing the value of the variable `resp` need no special treatment, so the above rules form a complete description, and a firm record, of our modeling choices. With these rules in place, the FeaVer model extractor generates the following model:

```
active proctype handshake()
{    int resp;

     p0!Msg;
     timer!Set(q0,16000);
     q0?resp;

     do
     :: c_expr {Ack == Phandshake->resp};
             timer!Reset(q0,0);
             break; goto C_0
     :: c_expr {TimeOut == Phandshake->resp};
C_0:         break; goto C_1
     :: else ->
C_1:         timer!Reset(q0,0);
             assert(false);
             break; goto C_2
     od;
C_2: skip;
}
```

The complete test harness definition for this example includes the test drivers, the channel declarations, and the five conversion rules. For large applications the test harness description is typically an order of magnitude or more smaller than the source text to which it is applied. Most important is, however, that it forms a complete and precise record of the test that is performed and the explicit assumptions that were made in setting it up.

The test harness provides a convenient point of control in the model checking process, providing a concise declaration and documentation of modeling and abstraction related choices. Once a test harness has been defined, it can be kept up to date with relatively little effort. Rather than, for instance, track differences between successive versions of the source text directly, FeaVer can track differences between completely populated conversion tables for each version (i.e., the user defined conversion tables to which the tool adds the implicit default conversion rules for each specific version of the source).

If code is merely is moved around within the source code, or if only the control flow changes,, there will be no differences in the conversion tables, and no action is required to repeat a verification. If code is added, it will show up in the difference as new default entries that may require new conversion rules. If code changes, some entries will disappear and others will appear. We have

developed a graphical user-interface for the model extractor that presents this difference information in an easily understandable format, so that a user can decide quickly what changes, if any, are needed before a system test is repeated.

6. Defining Logic Properties

The main purpose of constructing the framework for automated model extraction, based on highly condensed information specified in a verification test harness, is to simplify the model checking process for large software applications. The ultimate goal is to make it possible for any programmer to make effective use of model checking tools, without requiring detailed expertise in verification techniques or the use of logic. This immediately leads to the question how correctness requirements should be specified. So far, all logic model checkers work with specifications that are expressed in some flavor of logic: a temporal logic, such as LTL (Linear Temporal Logic) or CTL (Computations Tree Logic), or the mu-calculus, etc. These formalisms can sometimes be counter-intuitive, even for experienced users.

If we want the programming community at large to adopt model checking tools, it is important that there is a natural and intuitive way to specify correctness properties. To solve this problem, we can begin by looking at the way in which a limited form of program correctness properties are specified today, in the absence of model checking tools. Perhaps the best example is the use of the `assert(expr)` statement in C, which also appears in many other languages. Among programmers it is standard good programming practice to use assertions liberally in program source text [6,7,8]. Many programmers already follow this practice carefully in the development of their code, without prompting from program verifiers.

It is not too hard to instrument a model extractor to recognize these special statements, and to make sure that they are preserved in the generated models. The FeaVer tool goes a step further by also inserting additional assertions into a generated model, for instance to check that array indices are always within their declared bounds, and to check that pointers are always non-zero when referenced. But we can learn much more from the use of basic inline assertions.

FeaVer supports a small additional set of inline assertions that allows for slightly more sophisticated properties to be checked within a program source text. Two examples of such assertions are the `assert_p` precedence assertion and the `assert_r` response assertion. They are defined as follows:

```
assert_p(expr)
```

expresses the requirement that within a finite number of steps after the execution of this statement, the expression that is given as an argument must evaluate to true.

```
assert_r(expr1, expr)
```

expresses the same requirement, but additionally requires that `expr1` is currently true and remains true at least until `expr` becomes true.

These examples express simple temporal properties in a form that is readily understandable to any user, without prior exposure to model checking or temporal logics. Note that the first property is equivalent to the LTL formula:

```
[] (WhenAtL1 -> <> expr)
```

where the condition `WhenAtL1` is true whenever the executing process reaches the program location where the assert_p statement appears. Similarly, the second property would equate to the LTL formula:

```
[] (WhenAtL2 -> (expr1 U expr))
```

where the condition `WhenAtL2` is true whenever the executing process reaches the program location where the `assert_r` statement appears.

These two simple types of assertions can to some extent also be checked with automatically generated monitors that can be included in a conventional system execution, so their use is not purely limited to model checking applications. To make this possible for the `assert_p` primitive, for instance, the user may specify an upper-bound on the maximum duration of the interval within which the expression is required to become true. If the bound is exceeded without the target expression evaluating to true, a runtime error can be flagged.

8. In Conclusion

Possibly of greater importance to the success of a model checking effort than the selection of *the right tool* is the use of *the right model*. The model defines our main point of control in every verification effort, and must be chosen with care. Either too much or too little detail can cause either us, or our model checking tool, to lose track.

In this paper we have proposed that it can be advantageous to use automated model generation tools for model construction, driven by sets of rules. The rules can be defined and refined directly by a user, or by intermediate tools (e.g., predicate abstraction tools [9], program slicing tools, or theorem provers). In this way we can abstract almost completely from concerns about the representation of the control structure of an application: it can be automated. Instead, a model extractor allows us to

focus on the definition and maintenance of conversion or abstraction rules, which we propose is a simpler task.

We are beginning to gain some experience with this approach to software verification. We started in 1998 with the verification of the call processing code for a commercial telephone switch developed at Bell Labs [4]. Based on that experience, we generalized the approach from the use of a specialized format that had been adopted by the programmers of the call processing software for that switch, to unrestricted ANSI C-code. We named the resulting tool FeaVer in reference to its capabilities for software feature verification. This tool has been applied to a range of commercial software products, ranging in size from a few hundred to 160,000 lines of C source text. In all cases concurrency related errors were found in the source code that had escaped routine conventional system testing. The larger applications are of a size and a complexity that preclude the application of model checking based on manual model construction techniques.

The ideal, first advanced in the late sixties, of a formally justified, provably sound and complete methodology for software verification still stands, and is unchallenged by the methodology that is described here. There are many parameters in the construction of a verification test harness that remain subject to human judgment and are therefore also subject to human error. It would therefore be mistaken to compare this approach to a mathematical proof technique. In truth, the bar need not be raised quite this high for this approach to be considered successful. Software checking techniques of the type we have described compete more realistically with conventional software testing techniques. For concurrency related software defects, this methodology can significantly outperform that competition. In our application to the call processing software of a telephone switch at Lucent, for instance, we documented that the model extraction technique revealed ten times as many serious software defects in the target code than conventional system testing did. In a very real sense this allows us to produce more reliable systems software than before, which is after all one of the goals of software verification.

9. References

[1] T. Ball, and S.K. Rajamani. "Bebop: A Symbolic Model Checker for Boolean Programs," *Proc. SPIN 2000 Workshop on Model Checking of Software*, Lecture Notes in Computer Science, Vol. 1885, Springer Verlag, August/September 2000, pp. 113-130.

[2] J. Corbett, M. Dwyer, John Hatcliff et al. Bandera: Extracting Finite-state Models from Java Source Code. *Proc. Int. Conf. On Software Engineering*, ICSE, 2000, Limerick, Ireland, pp. 439-448.

[3] K. Havelund, and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer*, Vol. 2, No. 4, pp. 366-381.

[4] G.J. Holzmann, and M.H. Smith, "Automating software feature verification", *Bell Labs Technical Journal*, Vol. 5, No. 2, Murray Hill, NJ, April-June 2000, pp. 72-87.

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

[6] S.C. McConnell, "*Code Complete*," Microsoft Press, May 1993, 857 pgs.

[7] S. Maguire, "*Writing Solid Code*," Microsoft Press, May 1993, 256 pgs.

[8] D.S. Rosenblum, "A practical approach to programming with assertions," *IEEE Trans. on Software Engineering*, Vol. 21, No. 1, January 1995, pp. 19-31.

[9] 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, pp. 3-12.