

An Automated Verification Method for Distributed Systems Software Based on Model Extraction

Gerard J. Holzmann and Margaret H. Smith

Abstract—Software verification methods are used only sparingly in industrial software development today. The most successful methods are based on the use of model checking. There are, however, many hurdles to overcome before the use of model checking tools can truly become mainstream. To use a model checker, the user must first define a formal model of the application, and to do so requires specialized knowledge of both the application and of model checking techniques. For larger applications, the effort to manually construct a formal model can take a considerable investment of time and expertise, which can rarely be afforded. Worse, it is hard to secure that a manually constructed model can keep pace with the typical software application, as it evolves from the concept stage to the product stage. In this paper, we describe a verification method that requires far less specialized knowledge in model construction. It allows us to extract models mechanically from source code. The model construction process now becomes easily repeatable, as the application itself continues to evolve. Once the model is constructed, existing model checking techniques allow us to perform all checks in a mechanical fashion, achieving nearly complete automation. The level of thoroughness that can be achieved with this new type of software testing is significantly greater than for conventional techniques. We report on the application of this method in the verification of the call processing software for a new telephone switch that was recently developed at Lucent Technologies.

Index Terms—Formal methods, model checking, software verification, software testing, reactive systems, call processing, feature interaction, case studies.



1 INTRODUCTION

FEW programmers today would consider formal verification to be a serious option in their work, no matter how dedicated they are to producing reliable code. The main reason is the existing verification techniques usually require a considerable manual effort even by verification experts. Once the input to a verifier is fixed, the verification process itself can in many cases be automated, but the construction of the input (a faithful, verifiable, model of the software) requires specialized expertise and significant amounts of time. When formal verification is used, the required investment of time and expertise can often be afforded only once in a design cycle: at its start or at its completion. Verification is rarely used throughout a design, tracking its evolution, and intercepting bugs at the earliest possible moment.

The method described in this paper allows verification models to be extracted mechanically from the source of an application. A designer can now focus all attention on defining the precise type of verification to be performed. These definitions are captured in a “test harness,” which typically needs to be defined only once for a given type of application. The model extraction software can generate a sensible default for the test harness, that the user may adopt unchanged, or fine tune as the checking process advances.

With the test harness in place, the verification process can be mechanized from start to finish, even when the source of the application continues to change.

In principle, the method we describe here could be applied to software applications in a broad range of domains. The domain we are considering in this paper is that of event-driven (*reactive*) systems. Examples of event-driven systems include device drivers, distributed schedulers, concurrency control algorithms, and telecommunications applications. These systems are state-oriented. In each system state, one of a small number of known events is expected to occur. The system is designed to respond to the occurrence of such events in a well-defined way. A typical example of an event is the arrival of a message of a predefined type. The response can include the generation of new messages, to be sent to processes in the remote or local environment of the system. The specification of an event-driven system, then, includes definitions of sets of states, events, and actions, and it defines the various ways in which these elements can be combined.

The next section gives a simple example of an event-driven system of the type we will consider. The example specifies one possible implementation of the trivial alternating bit protocol [1]. Many types of flaws of distributed software applications of this type can be exposed mechanically with the verification technique we will describe. The scope of the verification based checks is substantially more thorough than what can be realized with conventional software testing techniques. We will say more about the nature of the remaining scope limitations in the conclusion of the paper.

• The authors are with Bell Laboratories, 2C-521 600 Mountain Ave., Murray Hill, NJ 07974. E-mail: {gerard, mhs}@research.bell-labs.com.

Manuscript received 4 Nov. 1999; revised 22 Feb. 2001; accepted 23 Apr. 2001.

Recommended for acceptance by S. Shatz.

For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 110899.

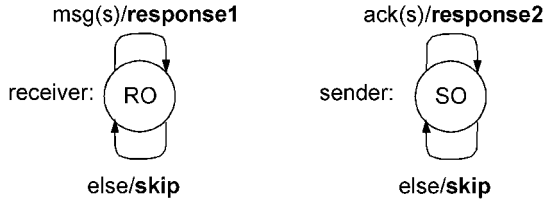


Fig. 1. A simple event-driven system.

1.1 Related Work

Earlier, several attempts were made to build translators that attempt to convert program text literally, without abstraction, into the input language of a model checker. Since no abstraction is used in these cases, one generally has to impose restrictions on the input language to keep the verification problem decidable. Such subsets were defined for Ada [11], Java [15], [16], and for SDL [18].

For Java, two recent projects are based on the systematic use of abstraction techniques: Bandera [9] and Pathfinder-2 [29]. The work in [9] bases the model extraction process on the use of program slicing techniques [15], [23], [28]. Both [9] and [29] also target the inclusion of mechanically verified predicate abstraction techniques by linking the model extraction process to either a theorem prover or a decision procedure for Pressburger arithmetic.

The work we present in this paper is based on the definition of an abstraction filter that encapsulates the abstraction rules for a given body of program code. The method is independent of the manner in which the abstraction rules are generated or justified. Importantly, they allow for the user to define or revise the rules either manually or mechanically.

A number of other techniques attempt to find a more informal compromise between conventional system testing on the one side and program verification on the other side. Examples include Verisoft [13], [14], which can be used on C code, and Eraser [27], which can be used for analyzing Java code. These tools analyze programs by monitoring a direct execution of the code. The checks are limited to basic safety properties, and do not permit the verification of general temporal properties. The method that we describe in this paper can be distinguished from these approaches by not only being more thorough (providing full temporal logic model checking capabilities), but also by allowing us to perform model checking on also partially specified systems.

1.2 Simple Example

In the example shown in Fig. 1, the behavior of two processes is defined: a sender and a receiver. The two processes have only one control state each in this case. In more typical applications, there may be hundreds of control states in each process definition. The single control state for the sender process is named $s0$ and the control state for the receiver process is named $R0$. The arrival of a message of type `msg` at the receiver constitutes an event. It is responded to with the execution of a piece of code that we have named **response1**. The second event specified here is the arrival of any other type of message, which is ignored. The fact that

such events are to be ignored is indicated here by the word **skip**. In both cases, control remains in state $s0$, although in general of course the processing of each event could bring the system into a different control state.

The sender process, on the right in Fig. 1, is responsible for sending messages of type `msg` to the receiver, but once a connection has been established it may only do so after the correct acknowledgment of the last transmitted message. Therefore, it should wait for the arrival of a message of type `ack` and responded to with the execution of a code fragment, named **response2**. All messages of other types are again ignored.

At least two types of messages are used in this protocol, `msg` and `ack`, and each is transmitted with a binary sequence number attached, which we have indicated here by writing `msg(s)` and `ack(s)`.

The code fragments that are executed provide the remaining information and they specify in particular how each process in this simple system can generate the events for the other. For simplicity, we will assume an infinite data stream to be transmitted from sender to receiver. The code fragments then look as follows: First, the receiver's response to the arrival of a message of type `msg` with sequence number s can be:

```
response1:
if (s == seqno) {
    accept_data();
    send(ack, s);
    seqno = 1 - seqno;
}
```

Any data carried in the message is only accepted if the sequence number parameter matches the local value of `seqno`, which toggles between zero and one. The sender's response to the arrival of a message of type `ack` with sequence number parameter s is:

```
response2:
if (s == seqno) {
    seqno = 1 - seqno;
    getnew_data();
}
send(msg, seqno);
```

If the acknowledgment carries the correct sequence number, the sender advances to the next message, otherwise it retransmits the old one.

An implementation of the control code for the receiver process could be as shown in Fig. 2, in ANSI C. The routine called `state()` is to be invoked (e.g., by the operating system) whenever an event occurrence has been detected. The invocation includes the type of the message received and its sequence number as parameters. In this code fragment, the current control state of the application is stored in a persistent integer variable called `state` and it can be modified as part of response processing for an event (though it remains unmodified in this example).

A control state marks a point in the code where system execution is to be suspended while the application waits for the occurrence of the next noteworthy event. An event may have to be responded to differently when received under

```

/* state names and event types: */
enum{ S0, R0, msg, ack, ... };

void
state(int event, int s)
{
    static int state = S0;
    static int seqno = 0;

    switch (state) {
    case S0:
        goto State_S0;
    default:
        report_error();
        return;
    }

State_S0: /* Receiver */
    switch (event) {
    case msg:
        response1 /* macro */
        state = S0; /* unchanged */
        break;
    default:
        /* no response */
        break;
    }
    return;
}

```

Fig. 2. C-code for receiver process.

different circumstances. Usually, there are multiple control states in a single application, defining possibly different responses to events.

Only some of the state information is encoded in the assignment of control states. Additional state information can be present in any persistent data that is used. In the example, such state information is present in the local copies of the sequence numbers that are maintained by sender and receiver. Selected data objects, then, can be important to the correct operation of the system, while the values of other types of data may be irrelevant to its basic operation (e.g., in this case, the detailed representation of the specific data stream that is transmitted). We should, therefore, be able to tell the verification system in a simple way what we consider to be directly relevant to the correct operation of the system, and what we can consider to be extraneous. The definition of what is relevant to a check is part of the construction of a test “harness.”

1.3 Outline of the Checking Process

Fig. 3 gives an outline of the checking process that we will discuss in more detail in the remainder of the paper. From a source specification, shown at the top of Fig. 3, a verification model is mechanically extracted in the target format of a logic verification system, with the help of a mapping table. The generated model can be considered a black box that the designer does not need to look at. The model is surrounded by test drivers.

A complete test harness consists of three parts. It contains

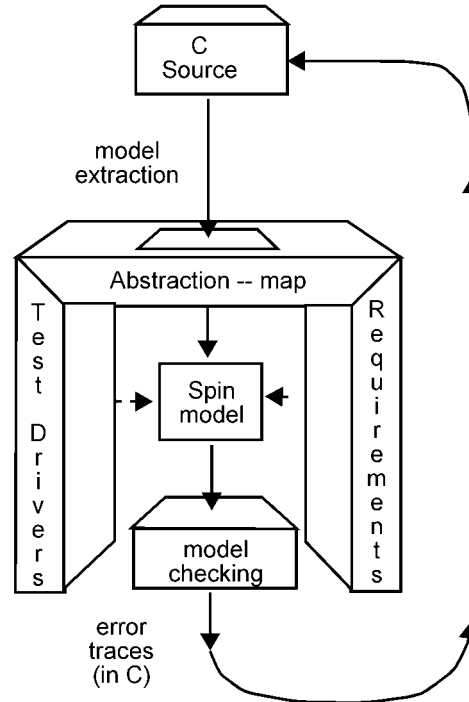


Fig. 3. Model extraction and test harness.

- a lookup table, or *map*, that defines which selected source statements and data manipulations are relevant to the tests to be performed,
- a *test driver*, that generates input to the system and possibly consumes its output, and
- a list of the *properties* to be checked.

Each of these three elements has a sensible default that makes it possible to perform a general verification of a software application from only generated code. The default lookup table, for instance, is simply empty, the default test driver randomly generates valid input messages, and when using the model checker **SPIN** [20] the default check is for absence of deadlock and unspecified reception errors (arrival of messages in states where no response is specified).

The implementation of the sender and the receiver processes from the example can be checked independently. The default for the sender’s *map* can be fine tuned, for instance, by marking statements of type `accept_data()` and `getnew_data()` as irrelevant to the check to be performed, and the other statements as relevant, by including them in the *map*. The test driver for the sender can be refined by using the specification of the receiver as a template, and vice versa. The set of properties to check for can be extended with a check for resilience to message loss, etc.

A model checker can reliably uncover sample execution traces of the violations of these properties that are hidden in the code as specified and it can report them at source level, that is, as traces of C statement executions, instead of as a trace through the model that was generated from the source.

Once the test harness is constructed, the remainder of the verification process can be performed without further user intervention, working from a database of previously defined properties to test for. It is possible, therefore, to define a system demon that watches the C code implementation of a system, awakens whenever that source has changed to repeat the verification of all relevant properties, and that emails any violations that are found to the last person who modified the code. The skills that are required to define a proper test harness are similar to the skills required to perform conventional software testing. The thoroughness of the tests performed, however, can be significantly greater.

1.4 Outline of the Paper

In Section 2, we describe what assumptions we make about the format in which the source of an event-driven system is specified. Section 3 describes how verification models are extracted from the code and Section 4 describes in more detail how a test harness is constructed. Section 5 discusses an application of the techniques outlined here to a commercial product. Section 6 gives an overview of a range of practical considerations in that have to be faced when attempting to perform model checking on applications of the size we are considering here. In Section 7, finally, we compare the approach with earlier efforts to apply formal verification methods in software design, and discuss how the scope of the new method compares with standard testing techniques.

2 SOURCE FORMAT

To extract verification models mechanically from source, the extractor must be able to reliably identify control states, event types, and the code fragments that constitute event responses. It takes remarkably little to accomplish this.

The extraction method that we will describe here accepts system specifications that are written in ANSI C, enhanced with one small extension. The extension itself was introduced independently of the desire to support a verification system: It was introduced by programmers to avoid the need to write repetitive code for implementing control states in event-driven systems. Specifically, the format was not introduced to support the verification attempt. The link to formal verification was made later, and had not originally be envisioned.

To illustrate the notation, which is informally known as the *@-format*, or more appropriately as *Thompson's format*, for its originator Ken Thompson, we give the matching revised implementation of the code for the `state()` routine of the sender process of the alternating bit protocol, discussed in Section 1.

```
/* state machine for Sender: */
void state(int event, int s)
{
    static int state = 1;
    static int seqno = 0;

    goto start;
@S0: /* control state */
    if (event == msg) {
```

```
        response1
    } /* else ignore */
    goto B_S0; /* state remains S0 */

start:
    switch(state) {
        @@ /* jump-table */
    }
    state = 0; /* missing state */
error:
    ...
out: /* exit label */
    return;
}
```

The code marked with @ is expanded by a small preprocessing filter into standard ANSI C code for the final implementation. For instance, the control state marker "@S0:" is expanded into:

```
B_S0:
    /* remember the state */
    state = 1;
    /* suspend the process */
    goto out;
A_S0:
    /* resume here - upon */
    /* next event occurrence */
    ;
```

The state name expands into two control points, as shown, named B_S0 and A_S0 (short for Before S0 and After S0, respectively). The first label corresponds to the point where the application is suspended while waiting for the next event occurrence. Jumping to this label causes the variable state to be updated with the numerical value assigned by the preprocessing filter, which is followed by a jump to the exit label out to suspend the process.

The jump table, at the label named start, expands into:

```
switch (state) {
    default: goto error;
    case 1: goto A_S0;
}
```

which upon each invocation of the routine (i.e., upon each event occurrence) restores the application to the correct control state and moves it to the point of execution that follows the suspension.

Not every control state needs to be named explicitly in this format. It is, for instance, possible to write a structured flow reflecting an expected sequential execution, as follows:

```
@idle:
    switch(event) {
    case A:
        action0;
    @:  switch(event) {
        case one:
            action1;
        @:  if (event != two) {
```

```

        goto error;
    }
    goto B_busy;
case B:
    goto A_idle;
}
break;
case B:
    action2;
    break;
}
goto B_ready;

```

There are three control states in this code, only one of which has been named explicitly. The preprocessing filter will assign internal names to the unnamed control states and arrange for the generation of all the standard code that is needed to create the state machine. In this case, the occurrence of event A in control state `idle` triggers the execution of code fragment `action0` followed by a process suspension, waiting for the next event occurrence. If the next event is B, we jump, without wait, to the corresponding case from the first control state, execute `action2` and suspend in control state `ready` (not shown here).

Especially in larger applications, the `@`-format can help to bring out the logic of a normal event flow through the system, which is much harder to achieve with a standard state machine format. The boiler plate that is needed to implement the underlying state machine is deemphasized, but can readily be reconstructed with a little preprocessor.

The use of the `@`-format allows us to easily extract a state machine model that can be used for formal verification from the source code of the system being designed. Although the use of the `@`-format simplifies the model extraction process, it should be noted that its use is not a precondition for this method. Models can also be extracted from C-code directly, from the abstract syntax tree generated in a standard C compiler [21].

3 MODEL EXTRACTION

Parsing a source file in `@`-format is fairly straightforward. The parser needs to recognize a range of C statements, including `switch` and `if` statements, but it needs to do very little interpretation of the text beyond being able to interpret event switches and the correct destination for each C `break` statement. The structure of the state machine can be issued in a range of formats, specifically it is readily converted into the format of a state based model checker such as **SPIN** [20].

3.1 Catch and Pry

The model extractor we have built consists of two programs, of about 1,500 lines of code each, called **pry** and **catch**. The first parses the application source and produces an intermediate, annotated, state machine format; the second reads in this intermediate format and generates the various pieces that make up the verification model. All C statements and expressions that appear in the source are tabulated by the parser. The source text for each such item is converted into canonical form, by removing all surrounding and included white space, and then looked up in a

mapping table to determine if the statement should become part of the model or can be ignored. If the statement does not appear in the map, i.e., if no explicit mapping for it was given by the user, the default is to consider it outside the scope of the verification. Happily, because of the way model checkers work, such declarations do not limit but *broaden* the scope of a verification attempt. For instance, if a condition is left unspecified, nondeterminism is introduced into the model and both possible truth values will be considered [20]. The user can chose to restrict the verification and make the models that are generated more specific, by editing the map.

Typically, the number of distinct types of statements and expressions used in an application is fairly small, which keeps even an exhaustive lookup table restricted to a few hundred entries for even a large source program. The parser that we have implemented warns the user for each statement that is outside the mapping table and for each entry in the mapping table that does not appear in the source. This level of warning has proven to be effective to alert the user when a new type of functionality was introduced into the application, or when an old functionality has disappeared. An update of the mapping table will suppress the warning and bring the verification model up to date with the revised version of the implementation. Importantly, updating the map is considerably simpler than updating the extracted model. The model extraction is mechanized, only the “rules” for the extraction process are maintained in the map. An example of the contents of a map follows in Section 4.

The details of the extracted model itself need be of little interest to a user. Used in verbose mode, our extractor inserts print statements into the model that reproduce the C source statements that cause the introduction of the various parts of the model. These print statements make it possible to convert any error scenario that the model checker finds (i.e., any violation of a stated property) into an execution trace through the C source code of the application itself. To interpret the error reports, then, no knowledge of the model structure is required, only knowledge of the application and of the properties that it was stated to satisfy in the test harness.

To allow for online verification of an evolving design, the user can concentrate all attention on the definition of the test harness that surrounds the modeled code.

4 TEST HARNESS

A test harness defines the scope and the bounds of a verification. It defines three basic elements:

- a *map* for selected source statements in the application,
- a (set of) *test drivers* that interact with the application, and
- a list of *properties* the application is required to satisfy.

We discuss these elements in more detail in the next three sections.

4.1 Statement Map

The statement map is used by the model extractor to define

- what is and what is not relevant to the checks being performed,
- which abstractions are used to perform the checks, and
- what representation is used for selected statements from the source in the target language of the model checker.

The map is stored as a text file with two columns: strings that identify C source statements appear on the left and the corresponding model fragments appear on the right. Any statement deemed irrelevant to the checks can be left out of the map, or it can be included with an explicit mapping to the Boolean value `true` for expressions, or `skip` for statements. (In **SPIN** `true` and `skip` are equivalent, which makes occasional mistakes harmless [20].)

As an example, a small part of the map that was used to verify the call processing code for a larger application discussed in Section 5 is as follows (line numbers added):

```

1 (x->drv->trunk)           false
2 !((x->drv->trunk))        true
3 fenable(x,CFBL)          true
4 !((fenable(x,CFBL)))     true
5 ((x->cswit&Swport)==0)    (cswit&Swport)
                           ==0
6 !((x->cswit&Swport))     !(cswit&Swport)
7 x->cswit&~Swport         cswit=
                           (cswit&~Swport)
8 x->cswit|=Swport         cswit=
                           (cswit|Swport)
9 x->drv->disconnect(x)     acs!Ctdis
10 x->drv->ring
   (x,Ctring,ipc)         acs!Ctring
11 memcpy                  true

```

The first two lines in this map define a restriction of the functionality of the model to cases where `nontrunk` calls are being processed, by forcing a specific test to evaluate to false and its negation to true. Because we are defining a verification model, the two evaluations need not be mutually exclusive. If we define, as in the next two lines, that a condition *and* its negation can evaluate to true, we state formally that both cases should be taken into account in the verifications to be performed, even though the detail of the condition itself (the function call) is completely removed from the model, being considered outside its scope. In this case, we define that a specific feature in the call processing code called CFBL (call forwarding busy line) can either be considered enabled or disabled. This introduces nondeterminism into the abstract model that conveniently broadens the scope of the verification.

Lines 5 and 6 define an explicit conversion of an expression from the source language into the modeling language. The mapping introduces a variable named `cswit` into the model and a constant named `Swport`. The manipulations of the structure element from the source are reproduced in the model (lines 7 and 8) with a simpler

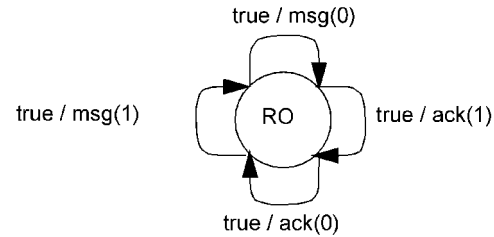


Fig. 4. Simple test driver for the sender from Fig. 1.

scalar variable. Lines 9 and 10 in this sample map show the abstraction of a function call into messages sent from the modeled application to its environment (during the checks: to test drivers), where it can trigger other events that may drive the application. In this case, it introduces a message channel named `acs` to which messages of type `Ctdis` and `Ctring` can be sent (our model checker uses a notation for message send and receive operations that is based on Hoare's CSP language [17]).

The last line in the map shows how the prefix of a statement can be used to identify a set of statements that should all map to `true` (and, hence, are abstracted from within the model). Here, any statement that performs a `memcpy` operation, irrespective of the arguments that are used, is mapped to `true`. The implicit match for the missing part of the statement will, of course, terminate at the first statement boundary.

4.2 Test Drivers

The environment stubs needed to perform verifications are similar in style and purpose to the test drivers that one needs to perform a conventional suite of software tests, cf. [22]. The main difference with conventional testing is that the environment stubs in the test harness of a verification model can generally be expressed more compactly, by exploiting the power of a nondeterministic specification. A small nondeterministic test driver can be functionally equivalent to a large set of deterministic test-drivers. For instance, a simple test driver for the sender process from Section 1 using nondeterminism to generate an infinite stream of `msg` and `ack` with sequence number equal to either 0 or 1, in arbitrary order, is illustrated in Fig. 4.

The sender should recognize only the `ack` messages with the proper alternation of sequence numbers from this stream of input. The verifier can detect if any other sequence of input might also be accepted, due to implementation faults.

A more detailed version of a test driver would behave more closely to the behavior that is specified for the receiver process, but it might also occasionally simulate the loss or duplication of messages, to test if the application can survive such errors, and continue to behave properly.

It may also be possible to produce more targeted default test drivers in a fully automatic fashion, with the aid of a static analysis of the source code text, as outlined in [15].

4.3 Default Properties

The default properties that a model checker such as **SPIN** can check for include absence of deadlock, completeness of the specification (e.g., the impossibility of message arrivals in control states where no response for the corresponding event has been specified), absence of nonprogress cycles, etc.

Especially for larger applications, the default checks prove to be quite effective in detecting basic flaws in the implementation of the state machines and uncovering unwarranted assumptions that are made about the behavior of other entities in an application's environment. It generally makes little sense to check an application for more detailed types of properties until the basic problems uncovered in the default checks have all been fixed. Once this point is reached, though, the model checker can offer a range of additional options to express detailed properties of the system in operation. Such properties are traditionally divided into invariant properties, called safety properties, that formalize specific global states of the system that should be unreachable, and dynamic, or liveness, properties that formalize requirements for either finite or cyclic execution sequences [20].

Basic correctness assertions can be added to the map or inserted into the test driver code. More sophisticated types of properties can be expressed in temporal logic (**SPIN** supports linear temporal logic for this purpose [25]), or they can be expressed as test automata, allowing for still greater control and expressiveness. In the next section, we give some examples of detailed properties that were culled from standards documents to test an implementation of a telephone call processing system designed at Bell Labs for Lucent Technologies.

5 INDUSTRIAL APPLICATION

The verification method we have described in this paper was developed for, and first applied in, the design and implementation of call processing software for a new Lucent Technologies network server product called PathStar™. The complete code for call processing is about 10,000 lines of sparsely commented C, approximately 10 percent of which defines the central state machine that drives the application. The state machine code written by Phil Winterbottom and Ken Thompson with the @ extension of C was the focus of the verification effort. No special accommodations, other than the format, were made in the code to facilitate the verification process.

The model extractor generates a **SPIN** model from the code in a fraction of a second, using a map file of roughly 250 entries. The map file was defined to provide complete coverage of all unique types of statements and expressions used in the state machine code, to make it easy to track changes in the evolving implementation throughout its development. We defined a series of test drivers that can simulate the behavior of subscribers placing originating and terminating calls, the behavior of various devices accessed by the software, and of internal watchdog timers. The complete description for these test drivers is approximately 450 lines of code in the format supported by **SPIN**. The size

of the mechanically generated model was approximately 2,600 lines of code. The original source code in C for the state machines roughly doubled in size during the period that we were tracking it with our verification test harness. After each update of the source, occasionally minor updates of the map were required before the verification suite could be repeated.

5.1 Property Definitions

For roughly the first half of the design period, the default completeness checks performed by the model checker were sufficient to uncover those aspects of the code that needed to be made more robust. Each error that was reported could automatically be rechecked after an update of the implementation, without user intervention. The test harness for such checks remained in place and each new version of the code could be tracked by extracting the new models mechanically from the implementation as explained. User intervention was restricted to updates of the statement map that could typically be made within minutes.

From the start of the verification project, we also set out to work on the definition of a comprehensive set of checks that the final design would have to pass in order to comply with existing recommendations for voice call processing and customer features. The reference for these properties was provided by Bellcore documents, indexed through [2].

Specific checks for roughly 20 separate features, such as call waiting, call forwarding, and three-way calling, were defined based on these documents, leading to the formalization of from five to 20 properties per feature. The higher numbers of properties correspond to multiparty calls, covering parties on hold, three-way calling, and call waiting. Many of the features also involve requirements on the specific ways in which multiple features should interact, e.g., through the enforcement of precedence relations. All known feature interaction requirements were captured in a database of properties that we assembled, accessible via a standard web-browser.

A complicating factor in this work was that the original requirements in the documents from [2] are specified informally in English prose. These informal requirements can be incomplete, contradictory, or express implementation bias, all of which complicate the checking process. We gave each relevant property a formalization in linear temporal logic (LTL) [25]. Encoding the properties in LTL allowed for concise documentation and a machine readable database and it may also serve to identify logical inconsistencies within the properties themselves, i.e., before the properties are used in the actual checks of an application.

5.2 Use of Templates

Many of the properties we considered conform to a small set of templates, or patterns [10]. Checks typically apply only within specific intervals, e.g., between an offhook and an onhook event from a subscriber to the phone system. Within the interval, we can check for required causal relations between specific trigger events and their associated response. Aspects of the check can also require certain global conditions to be true, e.g., that specific subscriber features be enabled or disabled within the interval of interest.

An example of a property is CFBL, Call Forwarding Busy Line. One of the requirements that we checked can be phrased informally as (cf. [3]):

If party S subscribes to CFBL, S is busy, and S receives an incoming call from party O, then the call from O to S will be forwarded.

The interval of interest here is the one where subscriber S is busy, e.g., while engaged in a call. The trigger event for the check is the reception of an incoming call request within this interval and the desired response is the forwarding of the call. Any other response from the state machine that controls subscriber S violates the requirement. Properties such as this can be hard to formalize. A safe method is to consult a reliable reference for standard types of properties. Most of the properties that we have encountered, for instance, appear in a patterns database for LTL that is being developed independently [10].

To formalize the property, we use five propositional symbols to capture the essential parts of the system state, as follows:

so: an originating call from subscriber S starts
 eo: the originating call ends
 si: an incoming call from subscriber O to S starts
 fw: the incoming call is forwarded by CFBL
 ei: the incoming call ends

and we can add some shorthands for combinations:

ni: (!eo /\ si)
 np: (!eo /\ !fw /\ !ei)
 ne: (!eo /\ !fw /\ ei)

The symbol ni captures the initiation of an incoming call while an outgoing call is in progress. Symbol np expresses the waiting for the incoming call to either terminate or be forwarded and ne captures the moment that the incoming call is terminated without having been forwarded, in violation of the CFBL requirement.

Potential violations of the property above can now be expressed formally in LTL as follows:

$$\langle \rangle (so \wedge X(!eo \cup (ni \wedge (np \cup ne)))) ,$$

where $\langle \rangle$, U, and X are LTL temporal operators (expressing respectively, eventually, until, and next) and where the symbols \wedge and $!$ are the standard operators for logical and logical negation. The formula states that it is a violation of the requirement if at some point in the execution (eventually operator) an outgoing call can start and following that (next operator) while the outgoing call is in progress an incoming call begins and can remain in progress until it terminates without being forwarded (until operator).

Formulae such as these can be converted mechanically into omega-automata [5] and then used in a model checking procedure using the algorithm outlined in [12]. The automaton will accept all those, and only those, execution sequences that correspond to a violation of the property. The model checker SPIN contains the conversion algorithm

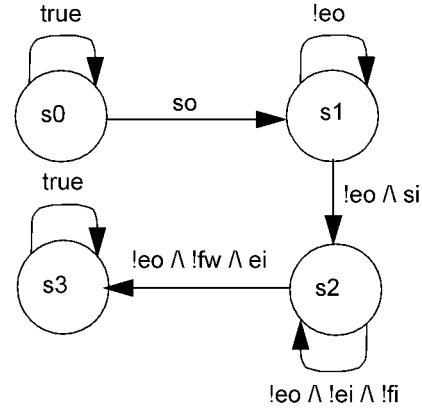


Fig. 5. Test omega-automaton for the CFBL feature (Call Forwarding Busy Line).

and can detect the violating sequences with a standard model checking run. Any violations that are detected can then be reported as execution traces through the original implementation source code of the application.

The test automata are also often simple enough that they can be constructed by hand and, in some cases, the hand-tuned automata can be smaller than the machine generated ones, which translates to reduced runtime requirements for the model checking process. The CFBL property discussed above, for instance, can be expressed in the four-state omega-automaton shown in Fig. 5.

The test automaton in Fig. 5 is designed to trap all executions that violate the property. This means that it need not be able to track any executions that *comply* with the property. For example, there is no transition in the automaton on state s2 for the occurrence of fw because this complies with rather than violates the property. For an execution sequence to be accepted in an omega-automaton, it is required that the automaton can return infinitely often to at least one of the accepting states (s2 and s3), [5].

The check of the property requires the CFBL feature to be enabled. We can achieve this by removing the possibility that the feature could be disabled from the statement map, that is by defining the negation of the Boolean condition `enable(CFBL)` to be false (cf. the sample map in Section 4).

This check was one of the tests performed by SPIN on an early version of the call processing code. After searching through a statespace of millions of reachable system states, SPIN reported an error trace that showed that the property was not necessarily satisfied under the stated conditions. The error, in this case, was not caused by the application but by an incompleteness of the *property* specification. The error trace, illustrated as a message sequence chart in Fig. 6, showed that an incoming call will not be forwarded if a feature named ACR, *anonymous call rejection*, is also enabled and the incoming call is from an anonymous source.

Fig. 6 shows one of the forms for a counter example that the model checker can produce as an alternative to direct C source execution traces. The subscriber (S) goes offhook, receives dialtone (indicated by a message to the subscriber's line equipment that turns on a digit recognizer), and dials

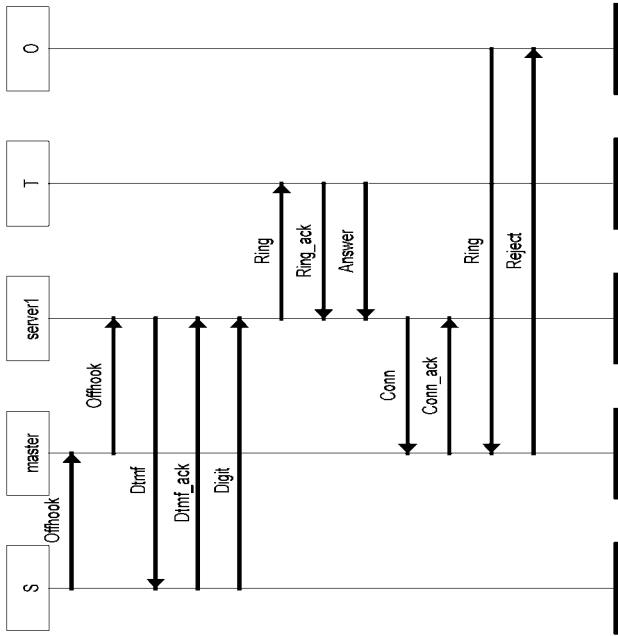


Fig. 6. Message sequence chart for a feature precedence error.

some digits to identify a remote subscriber T. The phone system requests a ringtone alert for the remote subscriber, which is acknowledged. The remote subscriber answers the call and a connection is established. At this point, another remote subscriber O places an incoming call to S, which is rejected despite the fact that we provisioned S with the CFBL feature. A trace like this can be reproduced with varying degrees of detail, as may be required by the developers to identify the circumstances under which this scenario can take place.

The existence of this error trace alerted us to the fact that an essential part of the feature properties was initially not captured in the test harness. To complete the definition, a precedence relation among subscriber features had to be added, which could then be used to verify that, for instance, CFBL is acted upon correctly in combination with specific other features that may or may not conflict with it.

If we do not update the property definitions, but directly repeat the test of the CFBL behavior with the ACR feature disabled, we find another error sequence. This time the counter example shows that, when the feature CW, i.e., call waiting, is enabled, call forwarding is also ineffective. The Bellcore recommendations [4] confirm that this is the way it should be. Next, also disabling the CW feature shows that the CFBL behavior cannot further be disrupted. In this case, the lack of a counter example, rather than its presence, is suspicious. Although the Bellcore recommendations do not state so explicitly, it is plausible that CFBL should also *not* be applied when a feature named DTS, Denial of Terminating Service, is enabled on a subscriber line. For our application, a targeted check, with just CFBL and DTS enabled, confirmed that the precedence setting was implemented incorrectly. After the correction of the code and the update of the properties for the inclusion of all explicit and deducible precedence relations, all checks could be repeated producing the required results. The value

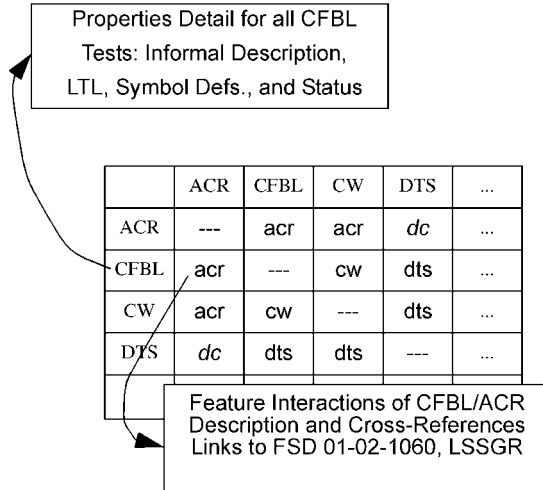


Fig. 7. Properties database, defining feature interactions and required feature precedence relations.

of a check of this type is that it conclusively demonstrates compliance with a stated property. The model checker does not test *some* selected execution scenarios, as a conventional testing tool may do, it checks *all possible* execution scenarios. Of course, doing so, it tries to do the least possible amount of work, using standard model checking optimization techniques [20].

What is illustrated by this example is that not every error results in an error scenario and not every error scenario corresponds to an error in the code. In all cases, however, the tester can state unambiguously what the outcome of each test ought to be, so that deviant result can be trapped as potential errors.

At the time of writing, roughly 25 implementation errors in the code have been intercepted by the model checking process we have described. Most of the errors found were deemed important, some critical. Most of these errors would have been virtually impossible to detect or reproduce with conventional testing techniques, due to the often subtle and hard to control timings that are needed to reveal them.

5.3 Property Database

All the information that is relevant to the checks performed is captured in a comprehensive database that can be accessed via a standard web-browser. Fig. 7 illustrates one of the information panels that is accessible in this way. Shown is the square matrix of all possible feature interactions, with all required precedence relations indicated. In this display, ACR, CFBL, CW, and DTS are feature acronyms, respectively for Anonymous Call Rejection, Call Forwarding Busy Line, Call Waiting, and Denied Terminating Service.

The database links each property to the original text of the underlying requirement in the Bellcore documents, with hypertext links. It also links to lookup tables that explain the naming conventions used and to status information on the checks that have been performed. The detail on the property descriptions themselves are stored as a row entry

Pending	Provisioning	Status	Action
$\exists(p \cup X(r \cup s))$	+TWC -DTS	Unchecked	Start Check
Running			
$\exists \langle \rightarrow er$	+TWC -DTS	B04331	Abort Run
Verified			
$\langle \rightarrow (a \wedge X(b \cup c))$ $\exists(p \rightarrow \langle \rightarrow q)$	+CFDA -ACR +CW +CFBL -ACR	B076443 M409574	Delete Result Delete Result
Failed		Error-Trail	
$\exists(oh \rightarrow \langle \rightarrow dt)$	-DOS +CFDA	A/50	Delete Result

Fig. 8. Web-based view of verification status.

in a separate table (cf. Fig. 8), specifying the variables needed to define the property.

The database is stored in a form that allows the properties to be read from the table by program and converted into the form that is needed to perform the corresponding model checking run. The properties can thus be included into the test harness without user intervention. If a property is violated, the model checker produces an error trace that is automatically linked back into the properties table as a bug report, for perusal and analysis by the user.

6 PRAGMATICAL CONSIDERATIONS

6.1 Search by Iterative Refinement

Short error traces are more convincing than long ones and, therefore, we rely on an option from our model checker to restrict the search to short error sequences. To find errors as quickly as possible, we also use a novel *iterative search refinement* procedure. A verification attempt starts with a coarse and, therefore, very fast approximation step. The precision of the search is increased step by step until either an error is located or exhaustive coverage is achieved.

This search method can be realized by exploiting a feature of **SPIN**'s bitstate search option (sometimes called *supertrace* [20]). By adjusting the size of the hash table, we can adjust both the speed and the coverage of a search. By doubling the size of the hash table after each search attempt, we can systematically increase the thoroughness of the search, and the maximum amount of time and memory that is consumed.

Errors typically show up in the first few iterations of this type of search, within the first few minutes. Only for properties that fail to generate counter examples, i.e., properties that are satisfied, we will go through the entire iteration process, from fast approximation down to a final and more time-consuming exhaustive check.

6.2 Keeping Track of Results

Fig. 8 gives a synopsis of the views offered by the web-based server for the verification status of properties. Each property in the database is in one of four possible states.

The property can be:

- **Pending**, when no verification results for this property have been obtained so far;
- **Running**, when a verification run is in progress;
- **Verified**, when no counter example to the property has been found in any of the verification runs performed;
- **Failed**, when at least one counter example to the property is available.

Depending on the state, an appropriate action can be performed on the property, directly through the web-browser interface. These actions are shown in the fourth column of the table in Fig. 8.

A verification can be initiated for a pending property, moving it to the running state by selecting the *Start Check* action from the table. A running property will typically transit quickly into either the *Failed* or the *Verified* state. Longer running verifications, however, can be queried about the intermediate results achieved thus far, by clicking on the run identifier that is displayed in the status field (the third column of the table). *Failed* and *Verified* properties can be reverted to a *Pending* state, by deleting the results from the table with the *Delete Result* action.

Both *Failed* and *Verified* properties can be queried about the nature of the result (a counter example or a coverage report) by inspecting the status report from the third column. We'll come back to the reasons for wanting to query the results of a *Verified* property below, in the subsection on *Avoiding False Positives*. We'll first look at the type of query we can perform for a *Failed* property.

6.3 Analyzing Counter Examples

As illustrated in Fig. 8, the first column of the results table gives the LTL form of a property. Symbols such as p , q , a , b , and c are defined by the user as part of the property definition. The detailed definitions for these symbols appear in the webpages in a list below the table (not shown here). The second column details provisioning data for a verification run. For the first verified property listed in Fig. 8, the provisioning data states that, for this property to be valid, we must assume specific feature combinations to be enabled or disabled, using feature acronyms. The meaning of the acronyms used here is as follows:

- TWC: *Three-Way Calling*,
- DTS: *Denied Terminating Service*,
- CFDA: *Call Forwarding on Don't Answer*,
- ACR: *Anonymous Call Rejection*,
- CW: *Call Waiting*,
- DOS: *Denied Originating Service*,
- CFBL: *Call Forwarding Busy Line*.

Features not mentioned in the table are either enabled or disabled by the verifier, nondeterministically. The third column links to the results of a verification, either the evidence for assuming that it is verified, or the counter example that shows that the property is not satisfied.

The identifier displayed in the third column indicates the type of verification that was performed (e.g., using the prefix E for exhaustive and A , B , or M for different types of approximate runs). In the case of a failed property, the length of the error trail is also encoded in the identifier (e.g., $A/50$ for a trail of 50 steps). The shortest error trail is always

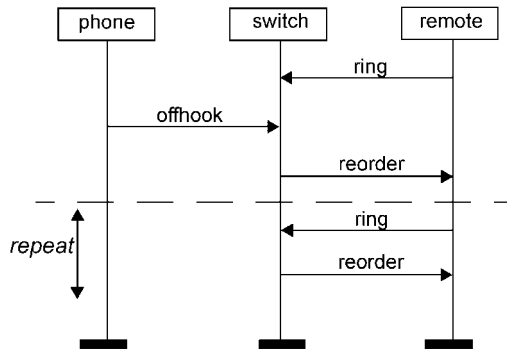


Fig. 9. Message sequence chart for a dial-tone generation error.

displayed first; if it is deleted, the next shortest trail is shown, etc. Clicking the hypertext link in the third column will display the detailed counter example for a failed property in one of four automatically generated forms:

- An ASCII representation of a message sequence chart showing only message send and receive operations between the concurrently executing processes, but not the details of statements executions in the individual processes.
- A graphical representation of the message sequence chart, adding some context information about the connection between the behavior specified in the property and the behavior exhibited by the system. To accomplish this the display notes when in the execution different predicates from the temporal logic formula change value.
- A statement-by-statement listing in C source code of the executions of all processes involved in the sequence.
- A statement-by-statement listing as above, with detailed dumps of all variable values at each step in the sequence.

Typically, the ASCII version of the message sequence chart serves as a quick indication of the type of execution that has been identified by the verification process and the C statement listings provide the finer detail that is needed to interpret it fully.

The last property included in the sample table shown in Fig. 8 states that if a subscriber has *CFDA* (call forwarding on don't answer) enabled, but not *DOS* (denial of originating service) then that subscriber will always receive a dialtone (represented by symbol *dt*) after going offhook (symbol *oh*).

When we verified this property for an early version of the call processing software, the verifier reported a short error sequence, which is illustrated as a message sequence chart in Fig. 9. When the PathStar system processes a call forwarding request for an incoming *ring* signal, it briefly enters a transit state. The processing in this state typically lasts for only a few milliseconds. The scenario, illustrated in Fig. 9, shows that if the subscriber happens to go *offhook* within that brief interval, it will not receive a dialtone. Normally, the system will timeout and generate a busy tone to alert the subscriber that no dialtone will be generated. The verifier discovered, however, that if there is a steady

stream of incoming *ring* requests, not even this busytone would be generated, resulting in a dead phone.

The error trace illustrated in Fig. 9 is an example of the type of software error that would be extraordinarily hard to identify with conventional testing, relying on delicate timings of events. Even if errors of this type are encountered in lab testing, they would be almost impossible to reproduce, and therefore often attributed to hardware glitches, beyond the control of the software.

6.4 Avoiding False Positives

Although the failed properties listed in the results tables give the most detailed feedback to a user, one quickly learns in an application of this size that also the purportedly verified properties can carry significant information. When a new property is formulated, it too has to be verified for its correctness. The first few attempts to capture a desired system property in a logic formula are often flawed. If the flawed property produces a false counter example, no harm is done. The counter example can be inspected and will reveal the flaw. If the verification fails to produce a counter example, however, it is not always evident if this is a vacuous result, e.g., because the property *accidentally* failed to match the intended behaviors.

In most cases, undesirable system behavior starts with a valid execution prefix that produces an invalid outcome. Consider, for instance, the setup of a three-way call. One of our formal properties for three-way calling checks that if the telephone subscriber follows all instructions for setting up the call correctly, that indeed a three-way call will be established. The valid execution prefix in this case is the subscriber correctly following the instructions. The invalid suffix of the execution would be the failure of the system to establish the call as instructed. If during the verification of this property neither the invalid suffix *nor* the valid prefix of the behavior can be matched, something is likely to be wrong with the formalization of the property itself.

The verification system that we have built gives the user feedback on the portion of the property formula that was matched in a verification run. The system provides this information in graphical form for all properties that are in the *Running* or in the *Verified* state. This means that, for a longer running verification, the user can see quickly if the verification attempt is progressing as intended and is likely to produce a useful result, or not. To achieve this, the system displays the test automaton that is derived by *SPIN* from the LTL property and indicates which states of this automaton have so far been reached in the verification attempt. This direct feedback has proven to be effective in diagnosing potential false positives.

7 THE TRAILBLAZER HARDWARE

After every update of the source code, all properties in our database are reverified from scratch. The iterative search refinement method secures that errors show up early in this process. Nonetheless, to run this process to completion, the verification of all properties in the database can take several hours. In addition to the iterative search refinement procedure outlined in the previous section, a number of other fairly simple optimizations can be made to speedup

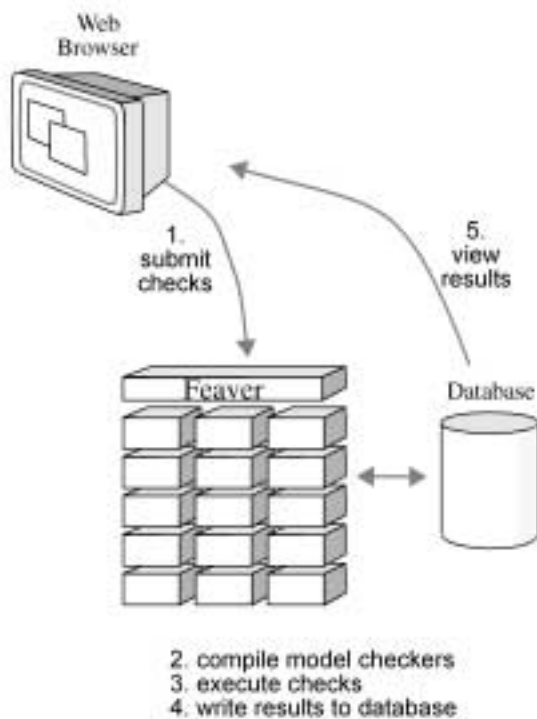


Fig. 10. The 16-CPU TrailBlazer hardware system.

this process. We discuss two of the optimizations that have had the largest impact on our project.

The first optimization we implemented exploits the way in which **SPIN** works. To perform a verification, **SPIN** always starts by generating C code that implements a model-specific verifier for the complete specification: one part in LTL and one part in **PROMELA**. In our case, the **PROMELA** model is extracted from the application, and therefore necessarily the same for all properties. Generating and compiling the complete verifier from scratch for each separate property is inherently wasteful. To prevent this, we have added an option that allows one to generate and compile the property specific code separately from the application code. This reduces the average compilation time for all properties from minutes to seconds. The large **PROMELA** specification is compiled into the model checker just once, and it is linked with a different small fragment of C code for each new property checked.

A second optimization exploits the fact that each property from our database is checked in a separate and independent verification run. It is easy to see that large gains in performance can be made if we parallelize these verifications. We have therefore built a dedicated 16-CPU system, called the *TrailBlazer* system, that performs all verifications. Each CPU has 512 MB of main memory and runs at 500 Mhz. The CPUs do not need disks since they are used only for computation. Each node of the system (illustrated in Fig. 10) runs a version of the Plan 9 operating system [24], called *Brazil*, as a compute server. The web-based interface runs on a separate system, named **FeaVer**, which also contains the central task scheduler. The scheduler is accessed through our web-based interface. It allocates verification tasks to the CPU servers and collects

the results of each run, storing it back into the web-based database of properties.

For the verification of individual properties, all levels of iteration can be run in parallel on this system. The verification process on all nodes is then stopped as soon as at least one of the nodes successfully generates a counter example to the given property. Using this type of parallel, iterative, search process, we have observed near interactive performance on verifications runs for the PathStar™ call processing source code. In effect, the TrailBlazer system provides the equivalent of an 8 GHz supercomputer.

8 CONCLUSIONS

In this paper, we have described a method that allows us to mechanically extract a model from a software implementation of an event-driven system, which can then be subjected to thorough verification with model checking techniques. The skills required to perform this type of check are not substantially different from those required to perform a conventional suite of software tests, although knowledge of model checking techniques can still be useful. Compared to testing, the same investment of time and energy can produce significantly greater rewards. Every test performed in this context will *prove*, instead of *probe*, full compliance with a given set of requirements.

Oddly, the method we describe is the reverse of one that is often advocated as the ideal design method for the application of formal methods. That ideal method starts with the design of an abstract model of the code, proves it correct, and then extracts an executable software implementation from that model, possibly using logically sound refinement techniques to secure that essential correctness properties are preserved. Much can be written about the relative benefits of that approach and the one we have described in this paper. We believe that the method to extract a verifiable model from evolving code has two advantages:

- It is easier to integrate with existing programming practice. Programmers need not be concerned about verification, or even be aware of it.
- It is easier to mechanize, making it possible to track an evolving design with little or no user intervention.

A mechanical method to extract verification models has a decisive advantage over methods that require a model to be constructed manually for a given software system, e.g., as described in [6], by avoiding reliance on expertise, and the investment of time by scarce model checking experts.

8.1 Limitations

The primary target of the method we have described is the verification of interaction problems in distributed software systems. This is the area where logic model checkers have shown their greatest benefits. By mechanizing the model extraction process we can more easily exploit this technology. The method is suited to find system deadlocks, race conditions, incompleteness, and the violation of any logic system property that is expressible in linear temporal logic. The method does not target and indeed would not be suited

for, the verification of computational aspects of sequential or distributed applications. This means that it would not be suited to use this method to verify, for instance, that a square-root computation always computes the correct result, or that a sorting algorithm always behaves as intended.

The scope of a verification of the type we have described is limited primarily by user-definable parameters that can be used as part of the test harness, e.g., the test drivers that feed input to the application, or the specific set of features that is allowed to be enabled for each test. This is similar to what one would encounter in traditional testing, but it deviates from the ideal of exhaustive logic program verification. For each parameter setting, though, the user obtains complete assurance that the model satisfies a property under the stated conditions. In that sense, the verification method described performs the equivalent of an exhaustive test, using optimized in-core model checking techniques. The model checker will generally run at a sustained rate where it tests millions of possible unique execution sequences per minute, carefully avoiding testing anything more than once using a range of search reduction techniques [20]. The properties that one can test for go well beyond what could be checked with a conventional type of test; ranging from plain assertions to subtle temporal statements about feasible or infeasible, possibly infinite, sequences of executions.

The construction of a test harness for a mechanized check along the lines we have described does require some specialized skills. We anticipate that more of the checking process can be automated. This holds in particular for the construction of the central mapping table, which may be simplified with the application of static analysis techniques and a range of predefined abstractions.

APPENDIX

THE MODEL CHECKING PROCESS

The model checker **SPIN** is based on an efficient implementation of an on-the-fly reachability algorithm for finite state systems that was first used in 1980 in a verification system called PAN (cf. [20]). In 1989, the descendant of that system was extended with an option for checking omega-regular properties (which includes linear temporal logic as a subset) and renamed **SPIN**. The sources to the **SPIN** system are distributed freely for educational and research purposes. The system is available via the web at <http://netlib.bell-labs.com/netlib/spin/whatispin.html>.

SPIN performs an efficient search of the reachable system state space to identify potential violations of user-specified correctness properties. **SPIN**'s input language is called **PROMELA**, which is short for Process Meta-Language. The language is designed to facilitate the concise, high-level, specification of distributed systems designs and contains direct support for nondeterminism. **SPIN** converts a **PROMELA** specification first into an automata model. From this automata model, it can compute a reduced composite state space. The logic property to be checked, specified as a temporal logic formula, is also converted into automaton form and checked against the composite state space. Any

execution sequence (path) in the composite state space that corresponds to the violation of a correctness property and can be reported as an error sequence.

The advantage of **SPIN**'s on-the-fly checking algorithm is that the critical checks for the presence of suspect execution paths in the composite state space can be performed even before the complete state space has been computed. Also, the check can be initiated on partially computed portions of that space. In practice, **SPIN** usually reports error sequences long before the computation of the composite state space would be completed. The complete state space is often only computed when the system to be checked is error free and no error sequences remain to be found.

More formally, each temporal logic property defines a language. This language formally describes all system executions that satisfy the property. If the property specifies a positive correctness requirement on system execution, we can negate it to obtain a requirement that formalises all the error sequences for that property. The system, formalised as a set of finite automata, also defines a language: the language of all the *feasible* system executions. **SPIN** computes the intersection of these two languages on-the-fly: the language of the negated property and the language of the system. If the intersection can be shown to be empty, the system is proven to satisfy the original property. If the intersection is nonempty, it contains the error sequences that demonstrate that the property is not satisfied. **SPIN** will report these error sequences to the user as counter examples to the correctness claim.

ACKNOWLEDGMENTS

The guidance and support of Ken Thompson and Phil Winterbottom have made the development and the application of this new verification method possible. Rob Pike and Jim McKie gave expert advice on how best to construct the TrailBlazer system.

REFERENCES

- [1] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson, "A Note on Reliable Full-Duplex Transmission over Half-Duplex Lines," *Comm. ACM*, vol. 12, no. 5, pp. 260-265, 1969.
- [2] Bellcore, *LSSGR, LATA Switching Systems Generic Requirements, FR-NWT-000064*, 1992 ed. Feature requirements, *SPCS Capabilities and Features*, Issue 1, Mar. 1996.
- [3] LSSGR, FSD 01-02-1450, p. 8.
- [4] LSSGR, FSD 01-02-1201, p. 2.
- [5] J.R. Buchi, "On a Decision Method in Restricted Second-Order Arithmetics," *Proc. Int'l Conf. Logic, Methods, and Philosophy of Science*, pp. 1-12, 1962.
- [6] W. Chan, R.J. Anderson, and P. Beame, "Model Checking Large Software Specifications," *IEEE Trans. Software Eng.*, vol. 24, no. 7, pp. 498-519, 1998.
- [7] J.A. Chaves, "Formal Methods at AT&T—An Industrial Usage Report," *Proc. Fourth Conf. Formal Description Techniques* pp. 83-90, 1992.
- [8] C. Colby, P. Godefroid, and L. Jagadeesan, "Automatically Closing Open Reactive Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 345-357, June 1998.
- [9] J. Corbett, M. Dwyer, et. al., "Bandera: Extracting Finite-State Models from Java Source Code," *Proc. Int'l Conf. Software Eng.*, 2000.
- [10] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett, "Property Specification Patterns for Finite-State Verification," *Proc. Second Workshop Formal Methods in Software Practice*, Mar. 1998.

- [11] M.B. Dwyer and C.S. Pasareanu, "Filter-Based Model Checking of Partial Systems," *Proc. ACM SIGSOFT Sixth Int'l Symp. Foundation of Software Eng.*, Nov. 1998.
- [12] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper, "Simple On-the-fly Automatic Verification of Linear Temporal Logic," *Proc. Conf. Protocol Specification, Testing, and Verification*, pp. 173-184, 1995.
- [13] P. Godefroid, "VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software," *Proc. Ninth Conf. Computer Aided Verification*, June 1997.
- [14] P. Godefroid, R.S. Hammer, and L. Jagadeesan, "Systematic Software Testing using Verisoft," *Bell Labs Technical J.*, vol. 3, no. 2, Apr.-June 1998.
- [15] J. Hatcliff, M.B. Dwyer, and H. Zheng, "Slicing Software for Model Construction," *J. Higher-Order and Symbolic Computation*, vol. 13, no. 4, pp. 315-353, Dec. 2000.
- [16] K. Havelund and T. Pressburger, "Model Checking Java Programs Using Java PathFinder," *Int'l J. Software Tools for Technology Transfer*, to appear 2000.
- [17] C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, vol. 21, no. 8, pp. 666-677, 1978.
- [18] G.J. Holzmann and J. Patti, "Validating SDL Specifications: An Experiment," *Proc. Conf. Protocol Specification, Testing, and Verification* pp. 317-326, June 1989.
- [19] G.J. Holzmann, "The Theory and Practice of a Formal Method: NewCoRe," *Proc. IFIP World Computer Congress*, vol. I, pp. 35-44, Aug. 1994.
- [20] G.J. Holzmann, "The Model Checker Spin," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279-295, May 1997.
- [21] G.J. Holzmann, "Logic Verification of ANSI-C Code with Spin," *Proc. Seventh Int'l SPIN Workshop Model Checking of Software*, Sept. 2000.
- [22] B. Marick, *The Craft of Software Testing*. Englewood Cliffs, N.J.: Prentice Hall, 1995.
- [23] L. Millett and T. Teitelbaum, "Slicing Promela and its Applications to Protocol Understanding and Analysis," *Proc. Fourth Spin Workshop*, Nov. 1998.
- [24] P. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," *Computing Systems*, vol. 8, no. 3, pp. 221-254, 1995.
- [25] A. Pnueli, "The Temporal Logic of Programs," *Proc. 18th IEEE Symp. Foundations of Computer Science*, pp. 46-57, 1977.
- [26] R. Saracco, J.R.W. Smith, and R. Reed, *Telecommunications Systems Engineering using SDL*. p. 632, North-Holland, 1989.
- [27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson, "Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs," *ACM Trans. Computer Systems* vol. 15, no. 4, pp. 391-411, Nov. 1997.
- [28] F. Tip, "A Survey of Program Slicing Techniques," *J. Programming Languages* vol. 3, no. 3, pp. 121-189, Sept. 1995.
- [29] W. Visser, S. Park, and J. Penix, "Applying Predicate Abstraction to Model Checking Object-Oriented Programs," *Proc. Third ACM SOGSOFT Workshop Formal Methods in Software Practice*, Aug. 2000.



Gerard J. Holzmann joined Bell Labs in 1980, after receiving the PhD degree in 1979 from Delft University of Technology in The Netherlands. He is a distinguished member of the technical staff in the Computing Principles Research group at Bell Laboratories. Dr. Holzmann designed and built a series of influential verification tools, leading in 1989 to the logic model checker named Spin. The Spin system is today considered one of the most widely used software verification tools. In 1998, Holzmann joined with Margaret Smith in the development of a new generation of tools for the direct verification of concurrent software applications written in ANSI-C, based on the paradigm of model extraction.



Margaret H. Smith received the MSc degree in industrial engineering in 1984 from the University of Michigan in Ann Arbor. She is a distinguished member of the technical staff in the Computing Principles Research Group at Bell Laboratories, where she works on application of computer-aided verification tools and on technology transfer.

▷ For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.