A Practical Method for Verifying Event-Driven Software

Gerard J. Holzmann
Bell Laboratories, 2C-521
600 Mountain Avenue
Murray Hill, NJ 07974 USA
+1 908 582 6335
gerard@research.bell-labs.com

Margaret H. Smith
Bell Laboratories, 2C-406
600 Mountain Avenue
Murray Hill, NJ 07974 USA
+1 908 582 5491
mhs@research.bell-labs.com

ABSTRACT
Formal verification methods are used only sparingly in software development. The most successful methods to date are based on the use of model checking tools. To use such tools, the user must first define a faithful abstraction of the application (the model), specify how the application interacts with its environment, and then formulate the properties that it should satisfy. Each step in this process can become an obstacle. To complete the verification process successfully often requires specialized knowledge of verification techniques and a considerable investment of time.

In this paper we describe a verification method that requires little or no specialized knowledge in model construction. It allows us to extract models mechanically from the source of software applications, securing accuracy. Interface definitions and property specifications have meaningful defaults that can be adjusted when the checking process becomes more refined. All checks can be executed mechanically, even when the application itself continues to evolve. Compared to conventional software testing, the thoroughness of a check of this type is unprecedented.

Keywords
Formal methods, model checking, software verification, software testing, reactive systems, telephone call processing, feature interaction, case studies.

1 INTRODUCTION

Few programmers would today 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 that the existing verification techniques usually require a considerable manual effort. 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, or *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 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 subsection 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]. The implementation we give is flawed, as many first-cut implementations might be. Virtually all flaws of applications of this type, however, can be exposed mechanically with the verification technique we will describe. The scope of the verification-based checks is substantially more thorough than can be realized with conventional software testing. We will say more about the nature of any remaining scope limitations in the conclusion of the paper.

Simple Example

In the example shown in Figure 1, the behavior of two processes is defined: a sender and a receiver.

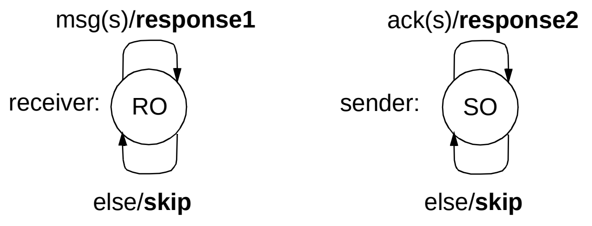


Fig. 1 — A Simple Event-Driven System.

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 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 Figure 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. It should therefore wait for the arrival of a message of type `ack` and respond to it 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 `s` 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 accepted only 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 sender process could be as shown in Figure 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 variable called `state`, and it can be modified as part of response processing for an event (though it remains unmodified in this example).

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

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

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

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

Fig. 2 — C-code for Sender Process.

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 different circumstances. Usually, therefore, 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.'

Outline of the Checking Process
Figure 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, at the top of Figure 3, a verification model is mechanically extracted, in the target format of a logic verification system. The model itself can be considered a black box that the designer does not need to look at. The model is surrounded (by the tester) with a test harness that captures the types of things we want to check the system for. This test harness consists of three parts. It contains

- a lookup table, or *map*, that defines which (types of) 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,
- 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 [14] the default check is for absence of deadlock and unspecified reception errors (arrival of messages in states where no response is specified).

When the lookup table is empty, the model extractor will produce an abstract (pure state machine) model that contains all control states and their associated event processing, but no data manipulation. Even at this level of abstraction, a model checker can identify subtle cases of incompleteness in an application. More precise types of checks become possible when the user starts populating the map.

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 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, repeats the verification of all relevant properties, and 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.

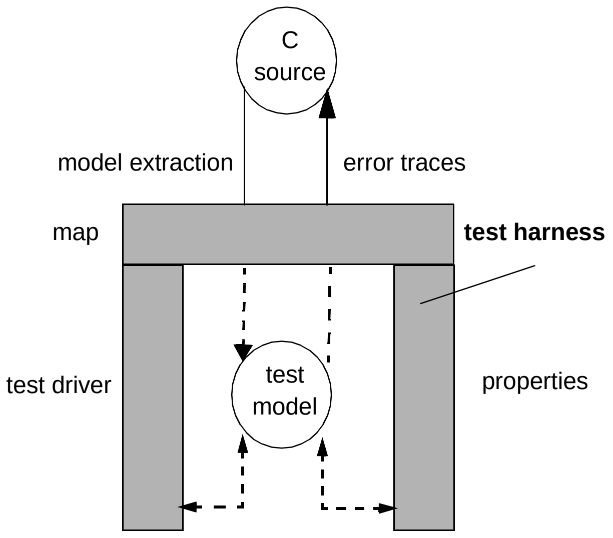


Fig. 3 — Model Extraction and Test Harness (shaded)

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. In Section 6, 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 we use 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 to avoid the need to write repetitive code for implementing control states in event-driven systems. The link to formal verification was made later, and had not originally been 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 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 pre-processing 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. 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, indicated by `@@` 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 possible, for instance, 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 pre-processing 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 helps to bring out the logic of a normal event flow through the system, which is difficult to achieve with a standard state machine format. The boiler-plate that is needed to implement the underlying state machine is de-emphasized, but can readily be reconstructed with a little preprocessor.

Importantly also, for the purposes of this paper, the @-format allows us to extract a state machine model that can be used for formal verification directly from the source code of the system being designed.

3 MODEL EXTRACTION

Parsing a source file in @-format is a fairly straightforward task. 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 is easily constructed and can be converted to other formats, specifically into the format of a state based model checker such as SPIN [14].

Pry and Catch

The model extractor we have built consists of two programs, of about 1500 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, non-determinism is introduced into the model and both possible truth values will be considered [14]. The user can choose 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 a software application is fairly small, which keeps even an exhaustive mapping table restricted to a few hundred entries for even a large 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. Updating the map is, of course, considerably simpler than updating a 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 correspond to 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 vio-lation 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 were stated 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 (types of) source statements,
- 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 sub-sections.

Statement Map

The statement map is used by the model extractor to define

- what is and what is not relevant to the checks being per-formed,
- which abstractions are used to perform the checks,
- 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 unique prefixes of C source statements appear on the left, 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 [14].)

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->cwswit&Swport)==0    (cwswit&Swport)==0
 6  !(x->cwswit&Swport)        !(cwswit&Swport)
 7  x->cwswit&=˜Swport  cwswit=(cwswit&˜Swport)
 8  x->cwswit|=Swport    cwswit=(cwswit|Swport)
 9  x->drv->disconnect(x)              acs!Ctdis
10  x->drv->ring(x,Ctring,ipc)        acs!Ctring
11  memcpy                                  skip
```

The first two lines in this map define a restriction of the functionality of the model to cases where non-trunk calls are being processed, by forcing a specific test to evaluate to false and its negation to true. Because we are defining a veri-fication 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 non-determinism into the abstract model that conveniently broadens the scope of the verification.

Lines 5 and 6 define an explicit mapping of an expression from the source text into a modeling equivalent. The mapping introduces a variable named `cwswit` into the model, and a constant named `Swport`. The manipulations of what is a structure element in the source are reproduced in the model (lines 7 and 8) with a 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 [12]).

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). Here any statement that performs a `memcpy` operation, irrespec-tive 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.

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. [15]. 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 non-deterministic specification. A small non-deterministic 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 non-determinism to generate an infinite stream of `msg` and `ack` with sequence number equal to either 0 or 1, in arbitrary order, is illustrated in Figure 4.

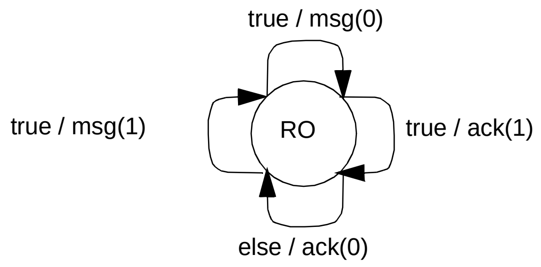


Fig. 4 — Simple Test Driver for the Sender from Fig. 1.

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.

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 non-progress 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 [14].

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 [16]), 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. The complete code for call processing is about 10,000 lines of sparsely commented C, approximately 10% 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 total of six test driver processes for this application, simulating 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.

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 5 to 20 properties per feature. The higher numbers of properties correspond to multi-party calls, involving 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 and made 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). Encoding the properties in LTL allowed for concise documentation and a machine readable database.

The database of properties could also serve to test other call processing applications, and it may also be used to identify logical inconsistencies within the properties themselves. We are considering tools that may be used for mechanically uncovering inconsistencies or redundancies within the set of properties, e.g., possibly with the use of theorem proving techniques.

Use of Templates
Many of the properties we considered conform to a small set of templates, or patterns [9]. 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 formalized as little test-automa that attempt to drive the application into executions that violate the requirements. Another safe method is to consult a reliable reference for standard types of properties expressed in a suitable logic. Most of the properties that we have encountered, for instance, appear in a patterns database for LTL that is being developed[1] independently [9].

Use of Logic [skip on a first reading]
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 moment that an incoming call is initiated 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:

```
<> (so /\ X(!eo U (ni/\(np U ne))))
```

where <>, U, and X are LTL temporal operators (expressing respectively, eventually, until, and next), and where the symbols /\, and ! are the standard operators for logical and, 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).

Test Automata
LTL formulae can be converted mechanically into test-automata [5] and then used in a model checking procedure, using the algorithm outlined in [10]. 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, 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 often also simple enough that they can be constructed by hand, and in some cases the hand-tuned automata are smaller than the machine generated ones, which translates to reduced run-time requirements for the model checking process. The CFBL property from above, for instance, is captured in the automaton from Figure 5.

1.The database is accessible via the web at http://www.cis.ksu.edu/˜dwyer/SPAT/ltl.html.

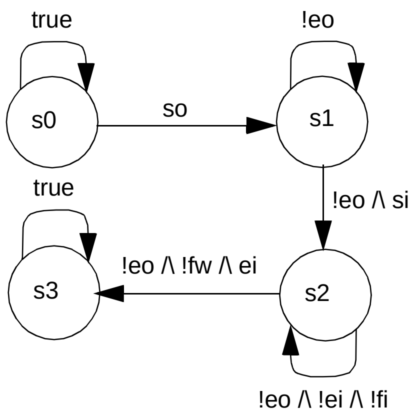


Fig. 5 — Test Omega-automaton for CFBL.

The test automaton in Figure 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, which we can do by removing the possibility that the feature could be disabled from the statement map, that is by defining the negation of the boolean condition `fenable(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 Figure 6, showed that an incoming call will not be forwarded if a feature named ACR, for *anonymous call rejection*, is also enabled, and the incoming call is from an anonymous source.

Figure 6 shows one of the forms for a counter-example that the model checker can produce, in addition to a C source execution trace that is the preferred form for the program developers. 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 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 has not been 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. Ignoring this observation, however, is still safe, but lengthens the checking process.

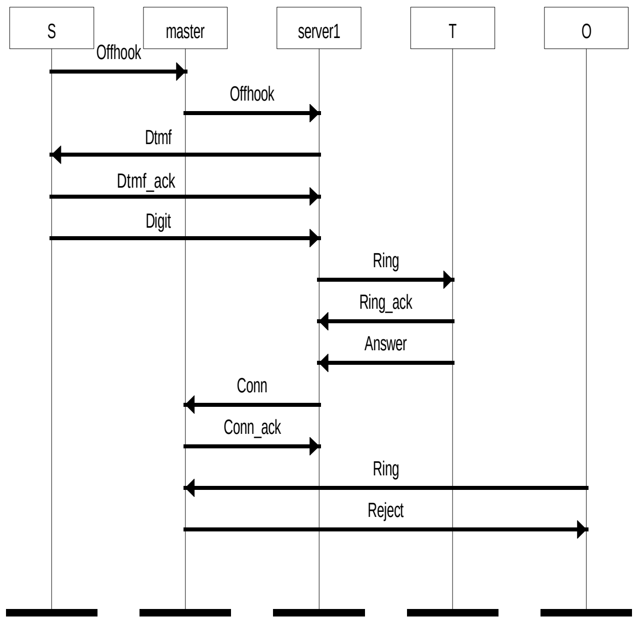


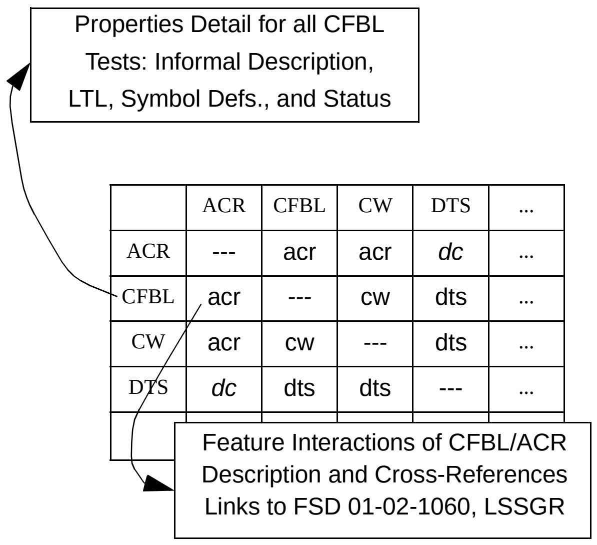
Fig. 6— Message Sequence Chart for an Error.

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, call waiting, is enabled call forwarding is also ineffective. The Bellcore recommendations [4] confirm that this is the way it should be. Next, disabling also 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 deducable precedence relations, all checks could be repeated producing the required results. The value 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 [14].

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.

Property Database
All the information that is relevant to the checks we performed is captured in a comprehensive web-based database. Figure 7 illustrates some of the information that is accessible through this database.


Properties Detail for all CFBL Tests: Informal Description, LTL, Symbol Defs., and Status
Feature Interactions of CFBL/ACR Description and Cross-References Links to FSD 01-02-1060, LSSGR

	ACR	CFBL	CW	DTS	...
ACR	---	acr	acr	*dc*	...
CFBL	acr	---	cw	dts	...
CW	acr	cw	---	dts	...
DTS	*dc*	dts	dts	---	...

Fig. 7— Properties Database, Defining Feature Interactions, and Required Feature Precedence Relations.

The database links each property to the original text of the underlying requirement in the Bellcore documents, it also links to lookup tables that explain the naming conventions that are used, and to current status information on the checks that have been performed. The detail on the property itself is stored as a row entry in a table, specifying the variables needed to define the LTL formula, or the template type that is used. For the template forms the tables also record the precise definitions of each propositional symbol used.

The database is stored in a form that allows each property to be read from the table and converted into the form that is needed to perform the corresponding model checking run, so that it can be inserted into the test harness without user intervention. If a property is violated, the model checker produces an error trace that can be linked into the properties table as a bug report to be inspected by the user.

6 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. With the same investment of time and energy, however, the rewards of the checks that are performed with a model checking tool can be significantly greater. 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 some of us have advocated as the ideal design method. 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 also has a decisive advantage over with 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 considerable investments of time and energy from model checking experts.

In an earlier effort to automate the model checking process, described in [7], [13], we built a system specification in a dialect of SDL [17] that we restricted to a verifiable subset. Although successful, the restrictions imposed by the verification system on the specification language prevented this approach from being applied more widely. In the current approach no significant restrictions are imposed on the source specification itself. Where desired, restrictions can be defined in the mapping table.

An interesting alternative approach to the one taken here was outlined in [11]. In this approach the source code of an application is instrumented to be run directly as part of the verification process. One adds test drivers to make the system self-contained, and places assertions into the source code. Choices inside the code can be made non-deterministic by further instrumenting the code. The verification system will in this case run the application through the choice points, attempting to cover as many distinct scenarios as is feasible within a given amount of time. Apart from message passing and the instrumented choices in the application code, there is no attempt to capture state information. The advantage is that only minimal work has to be done to run a test. Therefore, this method shares some objectives with the procedure we have outlined in the current paper. The new method described here goes further in providing a complete decoupling of the verification process from the application source code: the source code is never *modified* for a check. The code also does not need to be *executed* to perform a test. By executing the model instead of the code, the speed of a test can be increased by several orders of magnitude. The method described here, finally, provides greater control over the types of abstraction that are used, and should provide a significant increase in the scope and thoroughness of the tests that are performed.

The scope of a verification of the type we have described is limited only 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. 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 by using powerful search reduction techniques [14]. 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 default test drivers that our method generates could be improved further with the help of a static analysis of the extracted model, similar to what is described in, e.g., [8].

At the time of writing, January 1999, the application of the verification method we have described is still in progress, and potential extensions and refinements of the method are being studied. It should be possible, for instance, to apply the same methodology to the checking of general purpose software applications, outside the domain of event-driven systems we have discussed here.

ACKNOWLEDGEMENTS

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, Mihalis Yannakakis, Kedar Namjoshi, Stefan Leue, and Amy Felty provided valuable insights that helped to improve it.

REFERENCES

1. Bartlett, K.A., Scantlebury, R.A., and Wilkinson, P.T. ‘‘A note on reliable full-duplex transmission over half-duplex lines,’’ *Comm. of the ACM*, Vol. 12, No. 5, (1969), pp. 260-265.
2. Bellcore *LSSGR, LATA Switching Systems Generic Requirements*, FR-NWT-000064, 1992 edition. Feature requirements. Including: *SPCS Capabilities and Features*, SR-504, Issue 1, (March 1996).
3. LSSGR, FSD 01-02-1450, p. 8.
4. LSSGR, FSD 01-02-1201, p. 2.
5. Buchi, J.R., ‘‘On a decision method in restricted second-order arithmetics,’’ *Proc. 1960 Int. Cong. on Logic, Methods, and Philosophy of Science*, Stanford University Press, (1962), pp. 1-12.
6. Chan, W., Anderson, R.J., Beame, P., et al., ‘‘Model checking large software specifications,’’ *IEEE Trans. on Software Engineering*, Vol. 24, No. 7, (1998), pp. 498-519.
7. Chaves, J.A., ‘‘Formal Methods at AT&T - An Industrial Usage Report,’’ *Proc. 4th Conf. on Formal Description Techniques*, North-Holland, Amsterdam, (1992), pp. 83-90.
8. Colby, C., Godefroid, P., Jagadeesan, L., ‘‘Automatically closing open reactive programs,’’ *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Montreal, (June 1998), pp. 345-357.
9. Dwyer, M.B., Avrunin, G.S., Corbett, J.C., ‘‘Property Specification Patterns for Finite-state Verification,’’ *Proc. 2nd Workshop on Formal Methods in Software Practice,* Ft. Lauderdale, (March 1998).
10. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P., ‘‘Simple On-the-fly Automatic Verification of Linear Temporal Logic,’’ *Proc. Conf. on Protocol Specification, Testing and Verification,* Warsaw, Poland. Chapman & Hall, Germany, (1995), pp. 173-- 184.
11. Godefroid, P., ‘‘VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software,’’ *Proc. 9th Conf. on Computer Aided Verification,* Haifa, (June 1997). LNCS 1254, pp. 476-479, Springer-Verlag.
12. Hoare, C.A.R., ‘‘Communicating sequential processes,’’ *Comm. of the ACM*, Vol. 21, No. 8, (1978), pp. 666-677.
13. Holzmann, G.J., ‘‘The Theory and Practice of a Formal Method: NewCoRe,’’ *Proc. IFIP World Computer Congress,* Vol. I, Hamburg, Germany, (August 1994), pp. 35-44 North-Holland Publ.
14. Holzmann, G.J., ‘‘The model checker SPIN,’’ *IEEE Trans. on Software Engineering,* Vol. 23, No. 5, (1997), pp. 279-295.
15. Marick, B., *The Craft of Software Testing,* Prentice Hall, (1995), Englewood Cliffs, NJ, USA.
16. Pnueli, A., ‘‘The temporal logic of programs,’’ *Proc. 18th IEEE Symposium on Foundations of Computer Science*, Providence, R.I., (1977), pp. 46-57.
17. Saracco, R., Smith, J.R.W., Reed, R., *Telecommunications Systems Engineering using SDL,* North-Holland Publ, (1989), 632 pgs.

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. [14]). In 1989 the descendant of that system was extended with an option for checking omega-regular properties (including linear temporal logic properties as a subset) and was renamed SPIN. The source to the SPIN system is distributed freely for educational and research purposes. It is available via the web at http://netlib.bell-labs.com/netlib/spin/whatispin.html.

SPIN performs an efficient search of a subset of the reachable system state space to identify potential violations of user-specified correctness properties. SPIN's input language is called PROMELA, for Process Meta Language, and is designed to resemble mainstream programming languages, such as C, enhanced with constructs that facilitate the concise specification of non-determinism and message passing operations. Each process specified in PROMELA is converted into an automaton (an extended finite state machine). The model checker then computes a reduced composite state space from the process automata, which itself again corresponds to an automaton. The logic property to be checked, optionally specified as a temporal logic formula, is converted into another automaton, and then compared with the composite automaton for the system. Any system execution directly maps into an input sequences for the automata. Any input sequence that corresponds to the violation of a correctness property (specified by the test omega-automaton) that also appears in the composite state space automaton for the system, is an error-sequence that the model checker can report back to the user. The advantage of SPIN's on-the-fly checking algorithm is that the critical checks for 'overlap' between a test automaton and the composite system automaton can be done even before the complete system automaton has been computed: the check can already be done on partially computed fractions of the automata. In practice, SPIN usually reports the error sequences in a specification long before the complete composite automaton for the system would have been computed. The complete system automaton is typically only computed when the system is error-free, and no error sequences remain to be found.

More formally, each temporal logic property defines a language. This language contains all 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 formalizes all error sequences for that property. The system, formalized as a set of finite automata, also defines a language: the language of all feasible system executions. SPIN computes the intersection of these two languages: the language of the negated property and the language of the system. If the intersection is shown to be empty, the system has been proven to satisfy the original property. If the intersection is non-empty, it contains precisely the error sequences that demonstrate that the property is not satisfied SPIN will then select one of these error sequences and report it back to the user.