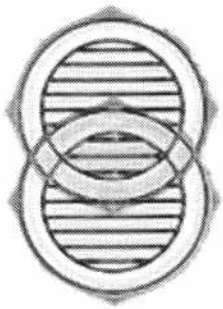
SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2001; **11**:65–79 (DOI: 10.1002/stvr.228)

Software model checking: extracting verification models from source code‡



Gerard J. Holzmann*,† and Margaret H. Smith

Bell Laboratories, Lucent Technologies, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

SUMMARY

To formally verify a large software application, the standard method is to invest a considerable amount of time and expertise into the manual construction of an abstract model, which is then analysed for its properties by either a mechanized or a human prover. There are two problems with this approach. The first problem is that this verification method can be no more reliable than the humans that perform the manual steps. If the average rate of error for human work is a function of the problem size, this holds not only for the construction of the original application, but also for the construction of the model. The standard verification trajectory therefore tends to become less reliable for larger applications. The second problem is one of timing and relevance. Software applications built by teams of programmers can change rapidly, often daily. Manually constructing an accurate abstraction of any one version of the application, though, can take weeks, which may jeopardize the validity of the results. In this paper a different verification method that avoids these problems is discussed. This method, which may be the precursor of a new class of testing techniques, was originally developed to allow for a thorough testing of parts of the software of a new commercial telephone switch. Here it is argued, though, that the method also has broad applicability to distributed software systems design in general.

KEY WORDS: software verification; model checking; model extraction; software testing; systems design; logic verification; automata; `Spin`

1. INTRODUCTION

In theory, software applications can be designed either top-down or bottom-up. It has been debated for decades which of these two methods would be better, when applied in isolation. In practice, the two methods are often used in combination. An industrial software design project normally starts with a

*Correspondence to: Gerard J. Holzmann, Bell Laboratories, Lucent Technologies, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

†E-mail: gerard@lucent.com

‡An earlier version of this paper appeared in the *Proceedings of FORTE XII / PSTV XIX*, which was published as: *Formal Methods for Protocol Engineering and Distributed Systems*, Wu J, Chanson ST, Gao Q (eds.). Kluwer, 1999; 481–499. It is reproduced here in modified form with the permission of Kluwer Academic Publishers.

Received 20 April 2000
Accepted 8 February 2001



top-down system design phase, where requirements and constraints are explored and defined. This is followed by a bottom-up phase, where developers try to meet the stated objectives. The design is then concluded with a system validation phase, where testers probe the code to check that the top-down and the bottom-up phases successfully meet in the middle.

If code were developed purely top-down, systematic design refinement techniques could be applied to guide the process, to ensure that each new level preserves all top-level requirements and continues to satisfy all constraints. This assumes, however, that the requirements and constraints do not change much, or that one would be willing to return to the first design phase each time they do change. It also assumes that one can successfully carry through refinement proofs from top-level design to implementation level code. At present, these assumptions are not very realistic for larger systems applications.

A verification problem is typically presented to a formal-methods person in the same way that it is presented to a software tester. In many cases a detailed implementation of the design already exists, and is available. The requirements for the design are often harder to find. Often, the requirements are not accurately documented and have to be reconstructed by the tester or verifier. If the problem is sufficiently important, a verification expert can undertake to formalize the requirements, and then to construct a formal model of the code that can be shown either to satisfy or to violate these requirements. Naturally, the formal-methods expert can make mistakes both in the formalization of the requirements and in the formalization of the code.

The effort to construct and to check the accuracy of a formal model for any application of substance can be significant, often consuming weeks or months. Only in rare cases, where design and development have ceased, and no further maintenance is performed, will the code remain unchanged in the period that is normally required to develop a satisfactory understanding of the system, to construct the formal models, and to perform the verification. This has several important implications.

- It is difficult to perform this type of verification during the design and development phases, when verification results are most beneficial. For this to be possible, the effort to construct an abstract model must be considerably smaller than the development effort itself.
- When verification results become available, it is probable that they apply only to older versions of the code, which may no longer be of interest to the designers or programmers.
- Manually constructed models can themselves be flawed, possibly hiding real errors and often introducing artificial ones. The latter can be detected and corrected; the former cannot.

An alternative method that will be explored in this paper is to extract a formal verification model mechanically from a software application. Once an initial framework for the extractions has been set up (defining abstractions, test objectives, and the equivalent of a test harness) the model extraction can often be performed in a fraction of a second, and verification can track design work virtually in real time.

2. MODEL EXTRACTION FROM CODE

For any software artifact there are at least two types of entities that are of interest when it comes to verification: the final version of the design requirements and the ultimate code that implements them. It is the task of the verifier to prove that the latter conforms to the former. There can be many hurdles to overcome before the verification itself can start.



First, the requirements may be partially unstated (i.e. incomplete); or they may be available only in their original form, before the long process is started in which idealized requirements meet real-world constraints and are 'true-ed up'. Ideally, the requirements are captured in a sufficiently formal notation, so that it will be possible to check them directly for their logical consistency, independently of the implementation. In practice, one often has to interpret and clarify informally-stated textual requirements.

A second hurdle to overcome is that the correctness properties of a system's implementation are often formally undecidable. For at least one well-defined subset of programs, however, decidability of any property that is expressible in linear temporal logic (a popular formalism for reasoning about distributed systems behaviour) can be guaranteed. All finite state programs are in that subset. The problem to be solved, then, is to devise a way to map a given program into an element of this subset by some well-defined process. The mapping process that will be used here is abstraction, and the result of this mapping is called a verification or a test model.

2.1. Verification during code development

A purely top-down design process would start with a high-level abstraction that can trivially be proven correct. By a disciplined process of refinement the designer would then try to obtain an implementation level description of the application that preserves the essence of the high-level abstraction. Specific types of functionality may need to be proven at each refinement level, in addition to the preservation of properties across levels. The process requires sufficient care and discipline that examples of its use in practice on large-scale applications are scarce.

The alternative that will be explored here is to allow the design and coding phases to proceed normally, including possible iterations, but to devise a method that can extract a verification model from the current version of the code at each point in time. The model can then be shown either to satisfy or to violate the current expression of all correctness requirements. Requirements and code are thus developed hand in hand. In some cases evidence of a violation of a requirement will necessitate a refinement of the requirements. In other cases it will call for an adjustment of the code.

2.2. Data and control

It will be assumed in what follows that the application describes a dynamic system of asynchronously executing processes, communicating via message passing and/or via access to shared data. That is, the verification of distributed systems software is specifically targeted for this work [1]. For each process there is a piece of code in the implementation that describes its behaviour. One would like to extract a model from each such piece of code, by applying carefully selected abstractions.

At the minimum, the control-structure has to be extracted from the code. Only a minimal parser is needed for the language in which the code is written to extract such a control-flow skeleton. For applications written in C, for instance, the parser needs to know that statements are terminated by semicolons, and it needs to know the syntax of selection and iteration constructs such as **if**, **while**, **until**, **goto**, **case** and **switch**, but it needs to know very little else about the language. In particular, the text of statements (assignments, conditions, function calls, etc.) can remain uninterpreted at this point.

The control-flow skeleton specifies only control; it is void of any knowledge of data so far. The skeleton can readily be converted into an initial verification model, e.g. for the model checker `Spin` [1].



Table I. Sample lookup table for defining simple abstractions.

Source code in C	Promela code for abstract model
`send(y,Iring)`	`y!Iring`
`send(x,Cranswer)`	`x!Cranswer`
`x->time = 60;`	`Time = true`
`cdr_init(y,ptr)`	`skip`
`printf(...`	`hide`

For an initial model, one can proceed by populating the skeleton model with all message-passing operations from the original code. To do so, one only has to be able to recognize which statements in the original source text in C correspond to `send`s and `receive`s, and to define a mapping for them into the skeleton model. The `send` and `receive` operations in the code can be more detailed than what is necessary or desirable in the verification model, so one can already choose to define a first coarse abstraction at this point.

In `Promela`, the specification language used by `Spin`, message `send` and `receive` operations are applied to channels (i.e. message queues). These operations typically include only those message parameters that are deemed relevant to the correctness properties of the application. Sequence numbers and congestion control information, for instance, would normally qualify for inclusion in such a mapping, but random user data would not. Could one automate abstractions of this type?

First note that both the source code and the code from the `Promela` model represent all message passing primitives in a specific, but distinct, notation. This consistent and context independent representation style in both the detailed and the abstracted version of the code allows for a particularly simple and elegant method of conversion. The translation can be accomplished with the help of a lookup table that is populated with the required conversions. The table contains the source code version of each primitive on the left, in a canonical version without white space, and the abstract version of the code that corresponds to it on the right, as illustrated in Table I. With this lookup table, the conversion from source code into a minimal model, containing just the representation of message-passing construct and control flow, can indeed be automated.

A critical observation at this point is, however, that the same discipline can be used to define an abstraction also for the remainder of all source statements. One need not limit oneself to just the message-passing primitives. The lookup table gives control over all aspects of the abstraction. It allows one to decide which data objects will be represented in the model, and how the manipulation of these data objects is to be rendered in the abstraction to secure a tractable model. Without the ability to define data abstractions, the model extraction process would be of no real practical significance. Consider, for instance, what useful properties would remain to be proven about a C program from which all data manipulations are removed.

If one wants to use the model extraction process that is sketched above during systems design, the source code may continue to evolve, also after the lookup table has been defined for an initial set of abstractions. New statements may be added to the code, old ones may disappear. Once the lookup table

is complete, though, it is relatively simple to keep it up to date. It is, for instance, considerably simpler than keeping a manually constructed abstract model up to date under the same conditions. To allow the tracking of changes in the source code of an application it suffices for the model extraction tool to alert the user to omissions, mismatches and redundancies that are detected between entries in the table and statements in the source code during the model extraction process.

The sample lookup table in Table I defines abstractions for five types of source statements that may appear in the source code of an application. In the first two cases, the mapping is a syntactical conversion from C to `Promela`. The third entry shows a more deliberate abstraction, where only the fact that a timer was set has been recorded, but not the numeric value to which it was set. The last two entries show a still greater abstraction. The fourth entry defines that all source statements of the type given on the left (having to do with issuing 'call data records' for billing in a telephone system) are irrelevant to the verification effort, and can be omitted from the model. The fifth and last entry specifies that all statements that start with the text string that precedes the three dots are to be suppressed from the model.

In the applications that the authors have studied for this method to date, there is on average one entry in the lookup table for every three to four lines of non-comment source text. The reason for this reduction in size is twofold. First, the same statements tend to reappear in different places in the code. The latter effect becomes more pronounced if abbreviations are used in the lookup tables, as illustrated by the last entry in Table I. Secondly, code that is used to indicate control flow alone does not show up in the tables at all. For a one thousand line program, therefore, a lookup table of roughly 300 entries can be expected. Introducing default translation rules for the most commonly occurring cases can reduce the table size still further.

The lookup table provides a convenient framework for working with abstractions that is independent of the specific way in which abstractions are chosen and justified. In many cases the derivation can be mechanized and separately proven to be logically sound, e.g. [2]. It is important to note that this framework supports, but intentionally does not require, formal proofs at each step. It thus allows also for more speculative systems exploration in a search for software flaws.

2.3. Model templates

The lookup table focuses on actions, not on the data objects acted upon. To produce the final model in `Promela`, also every abstract data object must be formally declared, defining its type and initial value. This can be done with the help of a model template that contains the required data declarations and an outline of the required process declarations, to be filled in with detailed behaviour specifications by the model extractor. The model template is constructed manually. It has a role in the verification process that is comparable to the role of a test harness definition in traditional testing.

A sample model template, of the type used for the case study discussed in Section 3, is shown in Figure 1. In this case the template is for a simple protocol system. The template defines the containers for two processes, and a number of local data objects that are used in the abstracted version of the code (not further detailed here). The model extractor replaces the placeholder `@P` with the abstracted behaviour that it generates from the source code of the application (not shown here).

Based on this notation, the authors are developing a more general methodology for defining test harnesses that supports the definition of abstractions and test architectures within a single format.



```
chan Q[2] = [1] of { mtype, bit };

active [2] proctype state()
{ mtype op, parl;
       bit seqno; byte who;
       chan qin, qout;

       who = _pid;
       qin = Q[_pid];
       qout = Q[1-_pid];

 @P   /* extracted code goes here */
}
```

Figure 1. A sample model template.

2.4. Checking abstractions

The adequacy of the abstractions defined manually in the lookup tables can be checked as follows. One can check, for instance, that all updates of data objects referenced at least once in the abstract model are represented in the abstraction. One can also check that all data dependency relations are properly preserved by the abstraction. If, for instance, data object **D** is part of the abstraction, one can check that all data objects on which **D** depends in the source code are also represented, and so on recursively. A check for these types of completeness can be automated trivially. One step further, one could also consider methods for mechanically deriving some or all of the lookup table contents from a given body of requirements.

2.5. Model extraction tools

Figure 2 gives an overview of a model extraction process of this type that the authors have implemented. Model extraction begins with a source description of an application in the programming language C. A modified C-parser, tentatively called AX (Automata eXtractor), extracts the control-flow graph from the C code. The program performs a data-dependency analysis, and uses the lookup table to generate the verification model, using the abstraction rules that are supplied in the lookup table.

Either manually or with the aid of other tools, the user constructs the lookup table, the model template, test drivers, and the properties to be verified. The test-drivers formalize assumptions that must be made about the environment in which the application will be executed. In the case of a telephone system, as discussed in the next section, the test drivers formalize assumptions about the behaviour of phone subscribers and of remote switches in the network, in so far as this is visible and relevant to the local switch. (The issue of visibility and relevance is discussed in more detail in [3].)

The user, then, provides only those parts of the model that define the desired level of abstraction, the properties that have to be proven, and the context in which they are to be proven, conforming to a familiar assume–guarantee paradigm. Most significantly, when the source code changes, only small

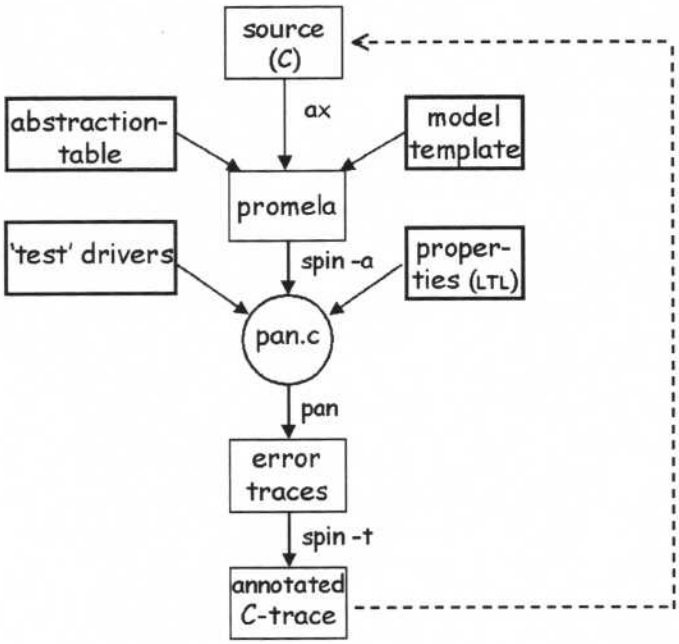


Figure 2. Overview of the model extraction process.

changes are needed before a verification attempt can be repeated. The required changes, made in the lookup table or in the model template, require no deep insights into the rationale for the changes in the source that produces the model. The model extractor issues warnings when it detects redundancy or incompleteness in the lookup table or model template.

A detailed description of the implementation of the model extractor for ANSI-C code can be found in [4].

3. APPLICATION: PATHSTAR®

The PathStar Access Server is a new Lucent Technologies product with support for traditional voice call processing, as well as data traffic in an IP network [5]. The system was designed and built by a large team of people over a period of several years. The call-processing software, one of many software modules, was checked with the methodology outlined here, and detailed in [6]. This application is, as far as the authors are aware, the first use of software model checking in a commercial product development of this size.



The call-processing code for PathStar was designed to adhere to the recommendations on call processing and feature processing published and maintained by Telcordia (the new name for the former Bellcore) [7], as is required for any public switching system. Structurally, however, the code differs from older implementations of the same functionality. The primary difference that concerns verification is that the central code related to call processing is concentrated in a single high-level procedure that can be studied and verified separately from the remainder of the code.

The central procedure encodes an extended finite-state machine that defines all control logic for telephony, i.e. basic call processing and all feature code. The code supports roughly fifty call-processing features, such as 'call waiting', 'call screening', 'call forwarding', 'three-way calling', etc. The more standard software modules with sequential code, e.g. defining the specifics of device drivers, switch administration, and operating systems interfaces, were tested separately, without use of model checking techniques.

As detailed in [6], the control logic for the PathStar system is written in standard C, more or less in state-machine format. (The structure of this code is irrelevant to the methodology described here, however.) From this code, the model extractor mechanically extracts the verification model, as illustrated in Figure 2. The correctness requirements were derived, by hand, from general Telcordia/Bellcore standards for call processing, feature behaviour, feature precedence relations, and from internal requirements on the desired functionality of the switch, that all switch vendors must adhere to.

The extraction process from C code to `Promela` code takes a fraction of a second. When a verification is repeated, minor changes in the source code require no human intervention. This applies in particular to small bug fixes that are meant to repair, for instance, correctness violations uncovered earlier by the verifier. Often the adequacy of a suggested fix can be confirmed conclusively within a few minutes of real time in this way. Within this framework it can be more time consuming to describe and record a bug in the standard project database than it is both to implement a correction and to verify its correctness.

More substantial changes to the source code, introducing new functionality, can require an update of the abstraction. To facilitate this process, the model extractor warns the user when it finds entries in the lookup table that do not (no longer) appear in the source code, and it warns when it finds new functionality in the source code that is undefined in the lookup table.

The evolution of the call-processing code for PathStar has been tracked from the first version, up to and including the final version of the code that was shipped to customers. All correctness properties were checked and rechecked for almost every version of the code throughout this period, often several times per day. In this period the code more than quadrupled in size. With few exceptions, each update of the lookup table took no more than five minutes. The few exceptions were when the source code changed more significantly from one day to the next, to support the inclusion of more complex feature code. In these cases the matching revisions of the lookup table took a few hours of edit time each. Compared to the time that is normally spent in coding and testing, these investments are modest.

3.1. Web-based access

The requirements database built by the authors to support the PathStar work contains over one hundred separate formalized requirements, and the results of the verification of each one, organized by feature.



A web server, named FeaVer (short for Feature Verification), gives online access to the various parts of this database, the outline of which follows.

- The text of the original Telcordia/Bellcore documents, in PDF format.
- Summaries of the verifiable parts of these documents, in ASCII format.
- Formalizations of each property in either temporal logic form, or in a shorthand form for specifying ω test automata. The shorthand was used to specify properties that involved long chains of events; normally requiring deeply nested `until` formulae in Linear Temporal Logic (LTL). The shorthand exploits the regular structure of such formulae, mildly increasing readability.
- The results of all verification attempts of the properties, in the form of tables derived directly from the machine generated results.
- Access to the reverification of individual properties, or groups of properties. To support this, each web page contains menus that allow a user to select either an individual property, or a group of properties, a verification mode, and a 'run' button that initiates the verification process. The results of the verification process are included in the web databases as soon as they become available, without further human intervention.

Each verification run initiated through these web pages always starts with the model extraction process, i.e. it starts with the call processing source code in C from the application itself.

The verification of a property stops when a counterexample is found, or when it completes normally. Because of the way that the generated models are instrumented, the execution of the counterexample with the `Spin` simulator suffices to generate an execution trace for the system in native C, i.e. as an interleaved sequence of C statement executions. The instrumentation applied is to allow the `Spin` simulation to print out the C source statements that correspond to the various portions of the abstract model, as comments to the simulation. To the C programmer, these comments are the equivalent of a true execution of the source code. If there is any discrepancy that makes this not true, it points to a flaw in the abstraction method applied, which is then repaired by a modification of the lookup tables that define the abstractions used.

3.1.1. Search by iterative refinement

Short error traces are more convincing than long ones, and therefore an option from the model checker `Spin` is relied upon to search deliberately for short error sequences. To find errors as quickly as possible, the system optionally uses an iterative search refinement procedure. If this option is chosen, a verification attempt starts with a coarse, and therefore fast, proof approximation step. This is realized by exploiting a feature of `Spin`'s bitstate search option (sometimes called 'supertrace' [1]). This algorithm is initially run with a deliberately small hash array. The supertrace algorithm performs the equivalent of a random pruning of the reachable state space, within the constraints imposed by the size of the hash array. The size of the hash array correlates directly with both speed and coverage.

By doubling the size of the hash array after each unsuccessful attempt the thoroughness of the search, and the amount of time spent performing it, can be systematically increased. Errors typically show up in the first few iterations of the search, within the first few minutes or seconds. Only properties that fail to generate any counterexamples, i.e. for properties that are likely to be satisfied, will go through the entire iteration process, from fast approximation down to a final thorough exhaustive check.



Table II. Sample web-based view of verification status. Abbreviations represent: TWC, three-way calling; DTS, denied terminating service; CFDA, call forwarding on don't answer; ACR, anonymous call rejected; CW, call waiting; CFBL, call forwarded busy line; DOS, denied originating source.

	Provisioning	Status	Action
Pending			
`[] (p U X(r U s))`	+TWC -DTS	n/a	Start check
Running			
`[]<>er`	+TWC -DTS	B04331	Abort run
Verified			
`<>(a ∧ X (b U c))`	+CFDA -ACR +CW	B076443	Delete result
`[] (p → <> q)`	+CFBL -ACR	M409574	Delete result
Failed		Error-trail	
`[] (oh → <> dt)`	-DOS +CFDA	B-50	Delete result

Table II gives an example of the view 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 properties are listed as formulae in LTL which are matched, on the web pages, by textual descriptions and clarifications.

A property in the database can be listed as:

- *pending*, when no verification results for this property have been obtained so far;
- *running*, when a verification run is in progress;
- *verified*, when no counterexample to the property has been found in any of the verification runs performed;
- *failed*, when at least one counterexample to the property is available.

Depending on the state, an appropriate action can be performed on the property. A verification can be initiated for a pending property, moving it to the running state. A running property will typically transit quickly into either a failed or a verified property. Longer-running verifications, however, can be queried about the intermediate results achieved thus far, by clicking on the status field (in the third column of the table). Both failed and verified results can be returned into pending properties, by deleting them from the table.

The most important feedback offered by this web-based tabulation of verification results is that all failed properties can be queried about the precise nature of the failure.

3.1.2. *Analysing counterexamples*

As illustrated in Table II, the first column of the table gives the LTL form of a property. Symbols such as `p`, `q`, `a`, `b`, and `c` are defined as part of the property definition. The detailed definitions for these symbols appear in the web pages 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 Table II, the provisioning

data states that for this property to be valid specific feature combinations must be assumed to be enabled or disabled. By default, each feature is nondeterministically enabled or disabled. The provisioning, if present, overrides this default by selecting one of the two possible states.

The third column links to the results of a verification, either the evidence for assuming that it is verified, or the counterexample that shows that the property is not satisfied.

The third column entry indicates the type of verification that was performed (exhaustive or approximate) and, in the case of a failed property, the length of the error trail. The shortest error trail is always displayed first; if it is deleted, the next shortest trail is shown, etc. Clicking the link in the third column will display the counterexample in one of four available forms.

- An ASCII representation of a message sequence chart derived from the trace. This chart shows only message `send` and `receive` operations between the concurrently executing processes, but no details of statements executions.
- A graphical representation of the message sequence chart, adding some context information about the connection between the behaviour specified in the property and the behaviour 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 Table II states that if a subscriber line has the feature CFDA (call forwarding on don't answer) enabled, but not DOS (denial of originating service) then the subscriber will always receive a dialtone (represented by symbol `dt`) after going offhook (symbol `oh`).

When this property was verified for an early version of the call-processing software, the verifier reported a short error sequence, which is illustrated in graphical form in Figure 3. When the PathStar system processes a call-forwarding request it briefly enters a transit state. The processing in this state typically lasts for only a few milliseconds. The scenario, illustrated in Figure 3, shows that if the subscriber happens to go offhook within that brief interval, no dialtone will be issued. As it turned out, in the original version of the code, the system would discard offhook signals while in this transit state and fail to exit the state unless the subscriber were onhook.

The error trace illustrated in Figure 3 is an example of the type of software error that would be extraordinarily hard to identify with conventional testing. Even if errors of this type are encountered in laboratory testing, they are almost impossible to reproduce, and are therefore often attributed to hardware glitches, beyond the control of the software.

3.1.3. 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.

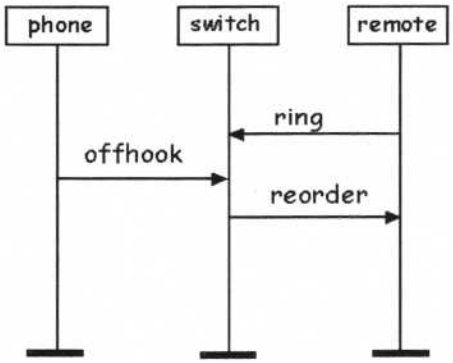


Figure 3. Message sequence chart for error sequence.

The first few attempts to capture a desired system property in a logic formula are often flawed. If the flawed property produces a false counterexample, no harm is done. The counterexample can be inspected and will reveal the flaw. If the verification fails to produce a counterexample, however, it is not always evident if this is a vacuous result, e.g. because the property accidentally failed to match the intended behaviours.

In most cases, undesirable system behaviour starts with a valid execution prefix that produces an invalid outcome. Consider, for instance, the setup of a three-way call. One of the formal properties for three-way calling checks that if the telephone subscriber follows all instructions correctly, then indeed a three-way call is 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 behaviour can be matched, something is likely to be wrong with the formalization of the property itself.

The verification system that has been 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.

3.2. The TrailBlazer system

After a full update of the source code, all properties in the database are reverified from scratch. The iterative search refinement method ensures 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.

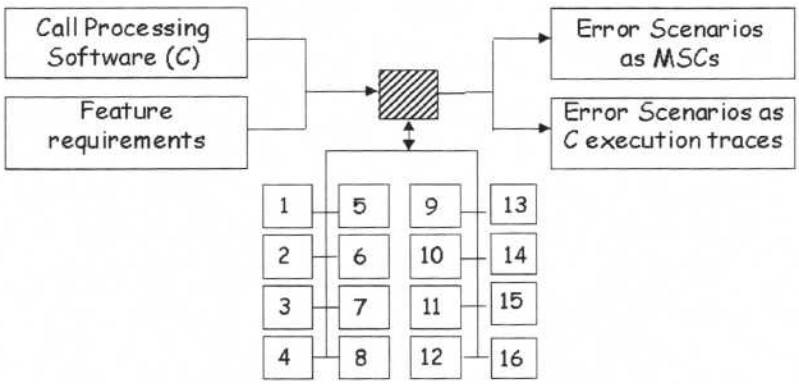


Figure 4. The TrailBlazer system, providing hardware support for FeaVer.

A number of simple optimizations can be made in this process. Two of the optimizations that have had the largest impact on the project will now be discussed.

The first optimization that has been 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`. Here, the `Promela` model is extracted from the application, and is 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, an option has been added to `Spin` 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 the database can be checked in a separate and independent verification run. It is easy to see that large gains in performance can be made if these verification jobs are parallelized. Therefore a 16-CPU system, called TrailBlazer, has been built to support the 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 Figure 4) runs a version of the Plan 9 operating system [8], Brazil, as a compute server. The web server runs on one of the 16 CPU servers and provides the link to the central task scheduler. This scheduler 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 counterexample to the given property. Using this type of parallel, iterative, search process, near interactive performance can be achieved on verifications runs, returning results to



the programmer within minutes after a revision is completed. In effect, the TrailBlazer system provides the equivalent of an 8 GHz supercomputer, built from fairly inexpensive off-the-shelf components.

4. CONCLUSIONS

A methodology has been sketched for software model checking that is based on an automated extraction of verification models from programs written in ANSI-C. Earlier attempts to verify C code have relied on manually constructed abstract models, requiring a considerable investment of time and expertise, e.g. [9]. To avoid this investment, one can try to avoid model construction and abstraction entirely, and allow a program to be run as is, sometimes under the control of a special scheduler [10], while monitoring selected operations for compliance with basic safety properties. If more complex types of properties or behaviours need to be checked, though, the use of abstraction techniques seems indispensable. The SLAM project represents C code as Boolean programs, abstracting the code much further than has been indicated here, thus trading some verifiability for generality [11]. Closer to the approach outlined in this paper, though more rigorous in checking the logical soundness of all abstractions made, are two projects that target the verification of Java code: Bandera [12], and Java Pathfinder [13].

The verification framework that has been described here was initially developed exclusively to allow a formalized check of the call-processing code for the new PathStar switch to be performed. As such, the methodology has proven to be remarkably successful [14]. It has intercepted many subtle bugs, without disrupting the normal development process. Many of the errors were deemed significant by the developers and unlikely to have been found by other means. Several of the errors uncovered in this way have led to a rethinking of the code structure itself, at an early state in its development, thus possibly avoiding more expensive redesign later.

On several occasions during the development of the code, the regular system testers independently discovered erroneous behaviour in the switch that could not be reproduced within their test framework. By specifying the pattern of events that had preceded each such observation in LTL, and then using a model checking run to generate matching executions, the model checker could be used to identify the timing conditions that could reproduce the types of failure that had been observed. With the help of the execution traces that the system generated, the cause of each such error could be diagnosed and repaired. The model checking framework thus proved its value not just as a verifier for abstract properties, but also as a diagnostic tool.

Using hardware support like TrailBlazer, a software tracking and debugging system has been built that allows developers to verify logical-correctness properties of source code with the thoroughness of a model checker, yet with the response time that is no longer than that of a traditional system test.

To improve the methodology further, the authors are exploring alternative methods for specifying software properties that are more intuitive than temporal logic. Testers are not used to defining logic formulae, but they have considerable experience in the construction of test drivers. A test driver can be seen as a special type of test automaton that is very similar to the automata that `Spin` extracts from an LTL formula. Allowing specific types of these automata to be constructed directly, perhaps guided by specially defined templates, comparable to those defined in [15], could help bridge the gap between testing and verification. The authors are also experimenting with simple graphical tools for visually

defining valid or invalid behaviour execution sequences. Such tools can again generate test automata, which can then be used in the verification process.

ACKNOWLEDGEMENTS

Phil Winterbottom and Ken Thompson inspired our work on the PathStar code, and provided valuable feedback and encouragement. Jim McKie and Rob Pike helped design and build the TrailBlazer system. We thank them all.

REFERENCES

1. Holzmann GJ. The model checker `Spin`. *IEEE Transactions on Software Engineering* 1997, **23**(5):279–295.
2. Dwyer MB, Hatcliff J, Joehanes R, Laubach S, Pasareanu C, Robby, Visser W, Zheng H. Tool-supported program abstraction for finite-state verification. *Technical Report*, Department of CIS, Kansas State University, August 2000.
3. Holzmann GJ. Designing executable abstractions. *Proceedings of Formal Methods in Software Practice*. ACM Press: Ft. Lauderdale, FL, U.S.A., 1998; 103–108.
4. Holzmann GJ. Logic verification of ANSI-C code with `Spin`. *Proceedings of SPIN 2000 Workshop* (*Lecture Notes in Computer Science*, vol. 1885), September 2000, Stanford University. Springer: Berlin, 2000; 131–147.
5. Fossaceca JM, Sandoz JD, Winterbottom P. The PathStar™ access server: facilitating carrier-scale packet telephony. *Bell Laboratories Technical Journal* 1998; **3**(4):86–102.
6. Holzmann GJ, Smith MH. A practical method for the verification of event driven systems. *Proceedings of International Conference on Software Engineering, ICSE99*, Los Angeles, May 1999; 597–607.
7. Telcordia/Bellcore. LATA Switching Systems Generic Requirements (LSSGR), *FR-NWT-000064*, 1992 edition. Feature requirements, including: *SPCS Capabilities and Features, SR-504*, Issue 1, March 1996.
8. Pike R, Presotto D, Dorward S, Flandrena B, Thompson K, Trickey H, Winterbottom P. Plan 9 from Bell Laboratories. *Computing Systems* 1995; **8**(3):221–254.
9. Chan W, Anderson RJ, Beame P, Burns S, Modugno F, Notkin D, Reese JD. Model checking large software specifications. *IEEE Transactions on Software Engineering* 1998; **24**(7):498–520.
10. Godefroid P, Hammer RS, Jagadeesan LJ. Systematic software testing using Verisoft. *Bell Laboratories Technical Journal* 1998; **3**(2):32–46.
11. Ball T, Rajamani SK. Bebop: a symbolic model checker for Boolean programs. *Proceedings of SPIN 2000 Workshop* (*Lecture Notes in Computer Science*, vol. 1885), September 2000, Stanford University. Springer: Berlin, 2000; 113–130.
12. Corbett J, Dwyer MB, Hatcliff J, Laubach S, Pasareanu CS, Robby, Zheng H. Bandera: extracting finite-state models from Java source code. *Proceedings of ICSE 2000*. ACM Press: Limerick, Ireland; 439–448.
13. Havelund K, Pressburger T. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer* 2000; **2**(4):366–381.
14. Holzmann GJ, Smith MH. Automating software feature verification. *Bell Laboratories Technical Journal* 2000; **5**(2):72–87.
15. Dwyer MB, Avrunin GS, Corbett JC. Property specification patterns for finite-state verification. *Proceedings of 2nd Workshop on Formal Methods in Software Practice*. ACM Press: Ft. Lauderdale, FL, U.S.A., 1998; 7–15.