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, USA

Key words: Software Verification, Model Checking, Model Extraction, Software Testing, Systems Design, Debugging, Call Processing, Telephony.

Abstract: 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 analyzed for its properties by either a mechanized or by a human prover. There are two main 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 rate of error for human work is a function of problem size, this holds not only for the construction of the original application, but also for the construction of the model. This means that the verification process tends to become unreliable 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 a faithful abstraction of any one version of the application, though, can take weeks or months. The results of a verification, then, can quickly become irrelevant to an ongoing design effort. In this paper we sketch a verification method that aims to avoid these problems. This method, based on automated model extraction, was first applied in the verification of the call processing software for a new Lucent Technologies' system called PathStar.™

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, applied in isolation, would be better. In practice, the two methods are often used in combination. An industrial software design project normally starts with a top-down system design phase, where requirements and constraints are explored and defined. This is followed by a bottom-up phase, where

J. Wu et al. (eds.), *Formal Methods for Protocol Engineering and Distributed Systems*

developers try to meet the stated objectives. The design is then concluded with a system validation phase, where testers probe the code to check if the top-down and the bottom-up phases meet in the middle.

If code were developed purely top-down, we could apply systematic design refinement techniques to guide the process, to secure 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 first design phase each time they do. It also assumes that we can successfully carry through refinement proofs from top-level design to implementation level code. At present, these assumptions are not very realistic for larger applications.

A verification problem is typically presented to a formal methods person in the same way that it is presented to a conventional software tester. In many cases a detailed implementation of the design 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 to either satisfy or to violate those requirements. 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 maintenance is performed, will the code remain unchanged in the period that model construction and verification is attempted. 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, chances are that they apply only to older versions of the code, which may no longer be of interest to the designers.
- Manually constructed models can themselves be flawed, possibly hiding real errors and often introducing artificial errors. The latter can be detected and corrected, the former cannot.

An alternative method that we will explore in this paper is to extract a formal verification model mechanically from a software application.

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 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, we often have to interpret and disambiguate informally stated textual requirements.

A second hurdle to overcome is that the correctness properties of the full implementation are almost certainly intractable. In general, only the correctness properties of a subset of all possible software applications can have decidable properties (cf. the general unsolvability of the halting problem). One such subset, of particular interest for verification, is the set of finite state programs. The hurdle to overcome, then, is to devise a way to map a given program into an element of this subset by some well-defined process. That well-defined process is called *abstraction*, and the result of this process can be called a *verification model*.

A purely top-down design process starts with a high-level abstraction that can trivially be proven correct. By a disciplined process of refinement the designer then tries 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 large-scale applications are scarce.

2.1 Verification During Code Development

The alternative that we will explore here is to allow the design and coding phases to proceed normally, 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 to either 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

We will assume in what follows that the application describes a dynamic system of asynchronously executed processes, communicating via message passing and/or via access to shared data. That is, we specifically target the verification of distributed systems software for this work [H97]. For each process there is a piece of code in the implementation that describes its behavior. We would like to extract a model from each such piece of code, by applying predefined, carefully selected, abstractions.

At the minimum, we have to extract the control-structure from the code. We only need a minimal parser 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 be converted into a minimal verification model, e.g. for the model checker SPIN [H97].

For a minimal model, we can proceed by populating the skeleton model with all message-passing operations from the original code. To do so, we only have to be able to recognize which statements in the original source text in C correspond to sends and receives, and to define a translation 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 model, so we can already chose to define a first 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 we 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. All that is needed is the definition of a lookup table that is populated with the required conversions. The table contains the source code version of each primitive on the left, and the abstract version of the code that corresponds to it on the right, as illustrated in Table 1 below. Once we have the 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 we can use the *same* discipline to define an abstraction for also the remainder of all source statements. We need not limit ourselves to just the message passing primitives. The lookup table gives us control over all aspects of the abstraction. It allows us 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.

Table 1. 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

If we want to use the model extraction process that is sketched above during systems design, the source code may continue to evolve, also after we defined the lookup table 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. Indeed, it is simpler than keeping a manually constructed abstract model up to date under the same conditions. To allow us to track the changes in the source code of an application it suffices for the model extraction tool to alert us to omissions, mismatches, and redundancies that are detected between entries in the table and statements in the source code during the model extraction process.

Table 1 gives a sample lookup table, defining abstractions for four 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 we record only the fact that a timer was set, but not the numeric value to which it was set. The last entry shows a still greater abstraction. It 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.

2.3 Model Templates

The lookup table focuses on actions, not on the data objects acted upon. To produce the final model in PROMELA, 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 behavior specifications by the model extractor.

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

active [2] proctype state()
{    mtype op, par1;
        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

A sample model template 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 behavior that it generates from the source code of the application (also not shown here).

2.4 Checking Abstractions

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

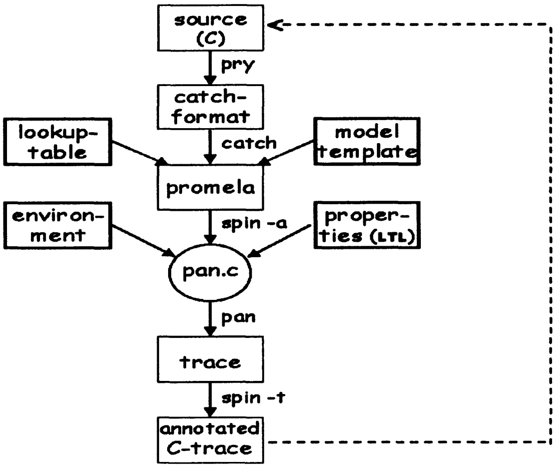

Figure 2. *Overview of the Model Extraction Process*

2.5 Model Extraction Tools

Figure 2 gives an overview of a model extraction process of this type that we have implemented. Model extraction begins with a source description of an application in the programming language C. A control-flow parser for a subset of C, called PRY, extracts an annotated control-flow skeleton from the C code, in an extended state machine format. As noted, PRY needs to identify only control-flow constructs and statement boundaries. It can therefore be substantially simpler and smaller than a full C compiler. In our implementation, for instance, PRY is a 1,700 line C program.

The next step in the model extraction process is performed by a tool called CATCH, of about the same size as PRY. CATCH reads the intermediate format produced by PRY, and guided by the lookup table and the model template it generates a verification model in PROMELA.

The user manually constructs the lookup table and the model template. The user also provides the properties to be verified and the test-drivers. 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 behavior of telephone 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 [H98)].)

The user, then, manually 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 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 tools PRY and CATCH issue warnings when they detect redundancy or incompleteness in the lookup table or model template.

3. APPLICATION: PATHSTAR™

The PathStar Access Server is a new Lucent Technologies product with support for call processing and telephone switching, as well as data connections, in an IP network [FSW98]. 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 in PathStar, was developed over a period of approximately nine months by Phil Winterbottom and Ken Thompson. The call processing software was verified with the methodology that we have outlined here, and detailed in [HS99]. This application is, as far as known to us, the first use of software model checking in a commercial product development at this scale.

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) [B92-96], as is required for any public switching system. Structurally, though, 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. At the time of writing, 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, are tested separately, without use of model checking techniques.

As detailed in [HS99], the control logic for the PathStar system is written in C, in a format devised by Ken Thompson to simplify the definition of the underlying state-machine. The code defines the conventional transition tuples of source state, guard condition, actions, and destination state. From this code we mechanically extract the verification models. The extraction process is performed in three steps, as also illustrated in Figure 2.

1. The C code for the core control code is parsed by the program PRY and converted into an intermediate format that makes both the state-machine structure and all transition tuples explicit.
2. The program CATCH reads this format and converts it into a standard PROMELA specification, using a model template and a lookup table (or *map*) to define the desired abstraction.
3. Correctness requirements, stated in linear temporal logic (LTL) are taken from a substantial database of properties that we have built derived from general Telcordia/Bellcore requirements for call processing, feature behavior, feature precedence relations, and from internal requirements on the desired functionality of the switch.

The complete extraction process takes a fraction of a second. Minor changes in the source code require no human intervention when a new series of verifications is performed. This applies in particular to small bug-fixes that are introduced to repair, for instance, a correctness violation uncovered by the verifier. Often the adequacy of a suggested fix can be (dis)proven within a few minutes of real-time, from start to finish. Within this framework it can be more time-consuming to describe and record a bug in the standard project databases then it is to both implement a correction and to formally prove its correctness.

More substantial changes to the source code, introducing new functionality, can require an update of the abstraction itself. This often means that the lookup table used by CATCH to guide the model extractions must be updated. To facilitate this process, the program CATCH 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.

We have tracked the evolution of the call processing code for PathStar 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 tripled 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.

3.1 Web-Based Access

The requirements database we have built 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 gives online access to the various parts of this database, containing:

- 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 short-hand form for specifying ω test automata (not further detailed here).
- The results of all verification attempts of the properties, in the form of tables derived directly from the machine generated results.
- Access to the re-verification 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 counter-example is found, or when it completes normally. Because of the way that we instrument the generated models, the execution of the counter-example 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 we apply is to allow the SPIN simulation to printout 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 we rely on an option from the model checker SPIN to deliberately search for short error sequences. To find errors as quickly as possible, our system optionally uses *an iterative search refinement* procedure. If this option is chose, 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* [H97]). We initially run this algorithm 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 we can systematically increase the thoroughness of the search, and the amount of time spent performing it. 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 counter-examples, i.e., for properties that are likely to be satisfied, we will go through the entire iteration process, from fast approximation down to a final thorough exhaustive check.

Table 2 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.

Table 2. Sample Web-Based View of Verification Status

Pending	***Provisioning***	***Status***	***Action***
[](p U X(r U s))	+TWC -DTS	n/a	Start Check

Running	***Provisioning***	***Status***	***Action***
[]<>er	+TWC -DTS	B04331	Abort Run

Verified	***Provisioning***	***Status***	***Action***
<>(a Λ X (b U c))	+CFDA -ACR +CW	B076443	Delete Result
[](p → <> q)	+CFBL -ACR	M409574	Delete Result

Failed	***Provisioning***	***Error-Trail***	***Action***
[](oh → <> dt)	-DOS +CFDA	B-50	Delete Result

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 counter-example to the property has been found in any of the verification runs performed;

  Failed, when at least one counter-example to the property is available.

Depending on the state, an appropriate action can be performed on the property. 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 Analyzing Counter-Examples

As illustrated in Table 2, the first column of the table gives the LTL form of a property. Symbols such as p, q, a, b, and c are defined by the user as part of the property definition. The detailed definitions for these symbols appear in the 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 2, the provisioning data states that for this property to be valid we must assume specific feature combinations to be enabled or disabled. (CFBL is *call forwarding busy line*, ACR is *anonymous call rejection*, and CW is *call waiting*). The third column links to the results of a verification, either the evidence for assuming that it is verified, or the counter-example that shows that the property is not satisfied.

The 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 counter-example 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 behavior specified in the property and the behavior exhibited by the system. To accomplish this the display notes when in the execution different predicates from the temporal logic formula change value.
- A statement-by-statement listing in C source code of the executions of all processes involved in the sequence.
- A statement-by-statement listing as above, with detailed dumps of all variable values at each step in the sequence.

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

The last property included in Table 2 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 we verified this property 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, it will not received dialtone. 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 would be onhook.

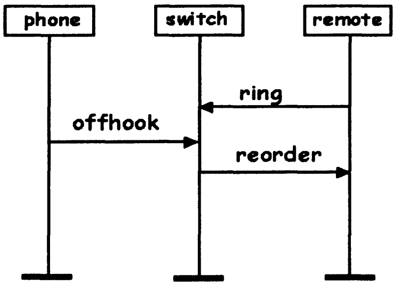

Figure 3. Message Sequence Chart for Error Sequence

The error-trace illustrated in Figure 3 is an example of the type of software error that would be extra-ordinarily hard to identify with conventional testing. Even if errors of this type are encountered in lab testing, they are almost impossible to reproduce, and 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. The first few attempts to capture a desired system property in a logic formula are often flawed. If the flawed property produces a false counter-example, no harm is done. The counter-example can be inspected and will reveal the flaw. If the verification fails to produce a counter-example, however, it is not always evident if this is a vacuous result, e.g., because the property *accidentally* failed to match the intended behaviors.

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

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

3.2 The TrailBlazer System

After a full update of the source code, all properties in our database are reverified from scratch. The iterative search refinement method secures that errors show up early in this process. Nonetheless, to run this process to completion, the verification of all properties in the database can take several hours. A number of simple optimizations can be made in this process. We discuss two of the optimizations that have had the largest impact on our project.

The first optimization we implemented exploits the way in which SPIN works. To perform a verification, SPIN always starts by generating C code that implements a model-specific verifier for the complete specification: one part in LTL and one part in PROMELA. In our case, the PROMELA model is extracted from the application, and therefore necessarily the same for all properties. Generating and compiling the complete verifier from scratch for each separate property is inherently wasteful. To prevent this, we have added an option 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 our database is checked in a separate and independent verification run. It is easy to see that large gains in performance can be made if we parallelize these verifications. We have therefore started the construction of a 16-CPU system, called the *TrailBlazer* system, that will perform the verifications. Each CPU is 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 [P95], called Brazil, as a compute server. The web-based interface runs on a separate system, which also contains the central task scheduler. The scheduler is accessed through our web-based interface. It allocates verification tasks to the CPU servers and collects the results of each run, storing it back into the web-based database of properties.

For the verification of individual properties, all levels of iteration can be run in parallel on this system. The verification process on all nodes is then stopped as soon as at least one of the nodes successfully generates a counter-example to the given property. Using this type of parallel, iterative, search process, we expect near interactive performance on verifications run through this system. In effect, the TrailBlazer system will provide the equivalent of an 8 GHz supercomputer.

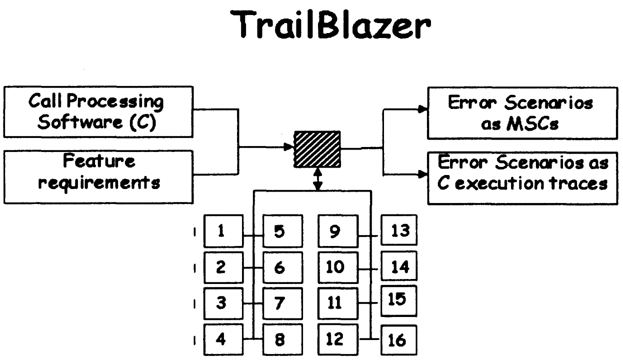

Figure 4. The TrailBlazer System

4. CONCLUSIONS

We have sketched a methodology for software model checking that is based on an automated extraction of verification models from source code. The techniques that we have implemented have proven to be effective, despite the simplicity of the means employed. The verification support we built for the PathStar system has to date identified nearly one hundred software bugs, without disrupting the normal development process. Many of the errors were deemed significant and unlikely to have been found by other means. Several of the errors uncovered in this way have lead 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 developers independently discovered erroneous behavior in the switch that could not be reproduced. By specifying the pattern of events that had preceded each such observation, we could use the model checker to identify the timing conditions that could reproduce the types of failure that had been observed. With the help of the error traces that our 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 the one planned for the TrailBlazer system, it should be possible to develop an intelligent software tracking and debugging system that allows developers to verify logical correctness properties of source code with the thoroughness of a model checker and the response-time of an ordinary software compiler. To make this possible, at least two issues must be addressed.

First, we would like to find an alternative for the specification of software properties that is more intuitive than temporal logic, yet equally robust. 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 [DAC98], could help us bridge the gap between testing and verification. A second issue that remains to be resolved is to examine if the method we have applied relies in any detail on the specifics of the PathStar application. We plan to look into this issue in detail by attempting to apply the method to a range of other software applications, adapting it where necessary.

ACKNOWLEDGEMENTS

Phil Winterbottom and Ken Thompson inspired our work on the PathStar system and helped to make it successful. Jim McKie and Rob Pike helped to design and build the TrailBlazer verification system. Kedar Namjoshi and Amy Felty helped with the formalization and the analysis of the correctness requirements for the PathStar application. We thank them all.

REFERENCES

[B92-96] LATA Switching Systems Generic Requirements (LSSGR), *FR-NWT-000064*, 1992 Edition. Feature requirements, including: *SPCS Capabilities and Features*, SR-504, Issue 1, March 1996. Telcordia/Bellcore.

[DAC98] 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*, March 1998, Ft. Lauderdale, Fl. USA, ACM Press.

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

[H98] Holzmann, G.J., Designing executable abstractions, *Proc. Formal Methods in Software Practice*, March 1998, Ft. Lauderdale, Fl., USA, ACM Press.

[HS99] Holzmann, G.J., and Smith, M.H, A practical method for the verification of event driven systems, *Proc. Int. Conf. on Software Engineering*, ICSE99, Los Angeles, May 1999.

[FSW98] Fossaceca, J.M., Sandoz, J.D., and Winterbottom, P., The PathStar™ access server: facilitating carrier-scale packet telephony. *Bell Labs Technical Journal*, Vol.3, No.4, Oct-Dec. 1998, pp. 86-102.

[P95] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., Winterbottom, P., Plan 9 from Bell Labs, *Computing Systems*, 1995, Vol. 8, No. 3, pp. 221-254.