Systematic Software Testing using VeriSoft: An Analysis of the 4ESS Heart-Beat Monitor

P. Godefroid, R. S. Hanmer and L. J. Jagadeesan

April-June 1998

Systematic Software Testing using VeriSoft: An Analysis of the 4ESS Heart-Beat Monitor

Patrice Godefroid
Bell Laboratories
Lucent Technologies
1000 E. Warrenville Road
Naperville, IL 60566, USA
god@bell-labs.com

Robert S. Hanmer
Lucent Technologies
2000 N. Naperville Road
Naperville, IL 60566, USA
hanmer@lucent.com

Lalita Jategaonkar Jagadeesan
Bell Laboratories
Lucent Technologies
1000 E. Warrenville Road
Naperville, IL 60566, USA
lalita@bell-labs.com

Abstract

VeriSoft is a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary code written in full-fledged programming languages such as C or C++. We report in this paper our analysis with VeriSoft of the 4ESS switch "Heart-Beat Monitor" (HBM), a telephone switching application developed at Lucent Technologies. The 4ESS HBM plays an important role in the routing of data in the switch, and can significantly impact switch performance. Since VeriSoft automatically generates, executes and evaluates thousands of tests per minute and has complete control over nondeterminism, our analysis revealed HBM behavior that is virtually impossible to detect or test in a traditional lab-testing environment. Specifically, we discovered flaws in the existing documentation on this application and unexpected behaviors in the software itself. These results are being used as the basis for an improved design of the HBM software in the 4ESS switch.

1 Introduction

Concurrent systems are systems composed of elements that can operate concurrently and communicate with each other. Each component can be viewed as a *reactive system*, i.e., a system that continuously interacts with its environment. Concurrent reactive systems are notably hard to design and test because their components may interact in many unexpected ways. Traditional testing techniques are of limited help since test coverage is bound to be only a minute fraction of the possible behaviors of the system. Furthermore, scenarios leading to errors are often extremely difficult to reproduce.

An effective approach for analyzing the correctness of a concurrent reactive system consists of *systematically exploring its state space*. The state space of a concurrent system is a directed graph that represents the combined behavior of all concurrent components in the system. For software systems, existing state-space exploration tools are restricted to the exploration of the state space of an *abstract description* of the system, specified in a *modeling language* (e.g., [HK90, Hol91, DDHY92, FGM+92, CPS93, McM93]). Several very complex examples of concurrent systems (e.g., communication protocols) have been modeled and then analyzed using such tools. In many cases, these analyses were able to reveal quite subtle design errors (e.g., [Rud92, CGH+93, BG96]). Once the model of a new software application has been thoroughly analyzed, it can also be used as the core of the implementation of the application (e.g., [HP89, FHS95]).

Recently [God97], it has been shown how systematic state-space exploration can be extended to deal directly with "actual" code implementing concurrent reactive software systems, in which processes execute arbitrary code written in general-purpose programming languages such as C or C++. *VeriSoft* is a tool for systematically and efficiently exploring the state spaces of such systems. By broadening the scope of systematic state-space exploration from modeling languages to programming languages, VeriSoft eliminates one major obstacle to a wider use of these techniques, namely the need to build a model of the software application to be analyzed.

In this paper, we report the first analysis of an actual software product using VeriSoft. We illustrate the application of VeriSoft to the analysis and re-engineering of the 4ESS switch "Heart-Beat Monitor" (HBM), an application that is part of the software controlling the 4ESS telephone switches developed by Lucent Technologies. The HBM of a 4ESS switch determines the status of different elements connected to the switch by measuring propagation delays of messages transmitted via these elements. This information plays an important role in the routing of data in the switch, and can significantly impact switch performance.

This paper is organized as follows. After a quick introduction to VeriSoft, we describe the Heart-Beat Monitor application, as well as the context and motivation for performing this detailed analysis. We then discuss the steps necessary to carry out the analysis of the HBM using VeriSoft. Because no modeling of the HBM code is necessary with VeriSoft, the total elapsed time before being able to run the first tests was on the order of a few hours, instead of the several days or weeks that would have been needed for the (error-prone) modeling phase required with traditional state-space exploration tools or theorem provers.

We then report the results of our analysis. Since VeriSoft automatically generates, executes and evaluates thousands of tests per minute and has complete control over nondeterminism, our analysis revealed unexpected behaviors of the HBM software that are virtually impossible to detect or reproduce in a traditional lab-testing environment. Specifically, we discovered flaws in the existing documentation on this application and unexpected behaviors in the software itself. Our results are being used as the basis for an improved design of the HBM software in the 4ESS switch.

In the light of these experiments, we conclude the paper with a discussion on the benefits and limitations of the new approach to concurrent program analysis that VeriSoft provides, and compare it with other approaches.

2 An Introduction to VeriSoft

2.1 Overview

VeriSoft [God97] is a tool for systematically testing concurrent systems composed of a finite set of *processes* and a finite set of *communication objects*. Each process executes a sequence of *operations*, that is described in a sequential program written in a full-fledged programming language such as C or C++. Such sequential programs are deterministic: every execution of the program on the same input data performs the same sequence of operations. We assume that processes communicate with each other by performing operations on communication objects, such as shared variables, semaphores, and FIFO buffers, for example. We assume that operations on communication objects are executed atomically: namely, no process can execute an operation on a given communication object while another process is currently doing so. The execution of an operation is said to be *blocking* if it cannot currently be completed; for example, waiting for the reception of a message blocks until a message is received. We assume that only operations on communication objects may be blocking.

We consider concurrent systems that are closed, i.e, that can evolve and change their state by executing operations. Thus, given a single "open" reactive system, the environment in which this system operates has to be represented, possibly using other processes, in order to close the system. For this purpose, a special operation "VS_toss" is available to express a valuable feature of modeling languages, not found in programming languages: *nondeterminism*. This operation takes as argument a positive integer n, and returns an integer in $[0, n]$. The operation is nondeterministic: the execution of VS_toss(n) may yield up to $n+1$ different successor states, corresponding to different values returned by VS_toss.

VeriSoft is a tool for systematically exploring the state spaces of closed concurrent (nondeterministic) systems. The state space of such a system is a directed graph representing all possible scenarios of operations that can be observed during any execution of the system. VeriSoft guarantees complete coverage of the state space up to some depth; hence, *all* possible executions of the system up to that depth are guaranteed to be covered. This is in contrast to conventional testing methods, in which only very few scenarios are explored. Also, VeriSoft has complete control over nondeterminism, and can completely reproduce any scenario leading to an error found during the search. More details about the state-space exploration techniques used by VeriSoft are given in the next section.

Two main classes of errors can be detected by VeriSoft: *deadlocks* and *assertion violations*. Deadlocks are states where the execution of the next operation of every process in the system is blocking. Deadlocks are a notorious problem in concurrent systems, and are extremely difficult to detect through conventional testing methods. Assertions can be specified by the user with the special operation "VS_assert". This operation can be inserted in the code of any process, and takes as its argument a boolean expression that can test and compare the value of variables and data structures local to the process. When "VS_assert(expression)" is executed, the expression is evaluated. If the expression evaluates to false, the assertion is said to be *violated*. Many undesirable system properties – such as unexpected message receptions, buffer overflows, and application-specific error conditions – can easily be expressed as assertion violations.

When an error of the type listed above is detected during state-space exploration, a scenario leading to the error state is exhibited to the user. An interactive graphical simulator/debugger is also available for replaying scenarios and following their executions at the instruction or procedure/function level (see Figure 1). Values of variables of each process can be examined interactively. Using the VeriSoft simulator, the user can also explore any path in the state space of the system with the same set of debugging tools.

2.2 State-Space Exploration in VeriSoft

In the VeriSoft terminology, operations on communication objects are called *visible operations*; VS_toss and VS_assert are also considered visible operations. The other operations are called *invisible*.

A concurrent system is said to be in a *global state* when the next operation to be executed by every process in the system is a visible operation. Every process in the system is always expected to eventually attempt execution of a visible operation. (When a process does not attempt to perform a visible operation within a given – user-specified – amount of time, VeriSoft reports an error called "divergence".) This assumption implies that initially, after the creation of all the processes of the system, the system may reach a first and unique global state s_0, called the *initial global state* of the system. A *transition* is one visible operation followed by a finite sequence of invisible operations performed by a single process and ending just before a visible operation. The *state space* of the concurrent system is then defined as the global states that are reachable from the initial global state s_0, and of the transitions that are possible between these. It can

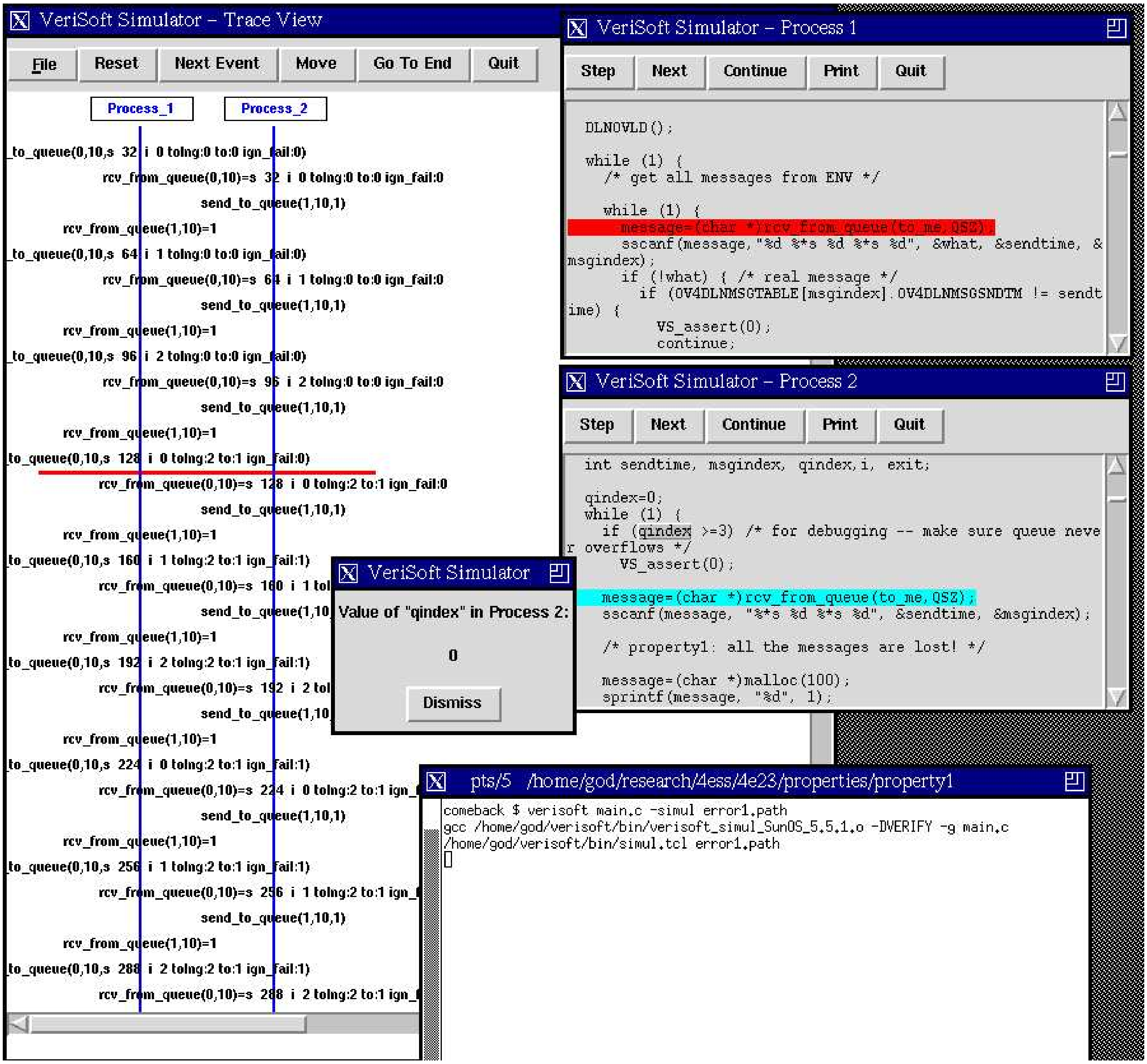

Figure 1: Screenshot of the VeriSoft Simulator

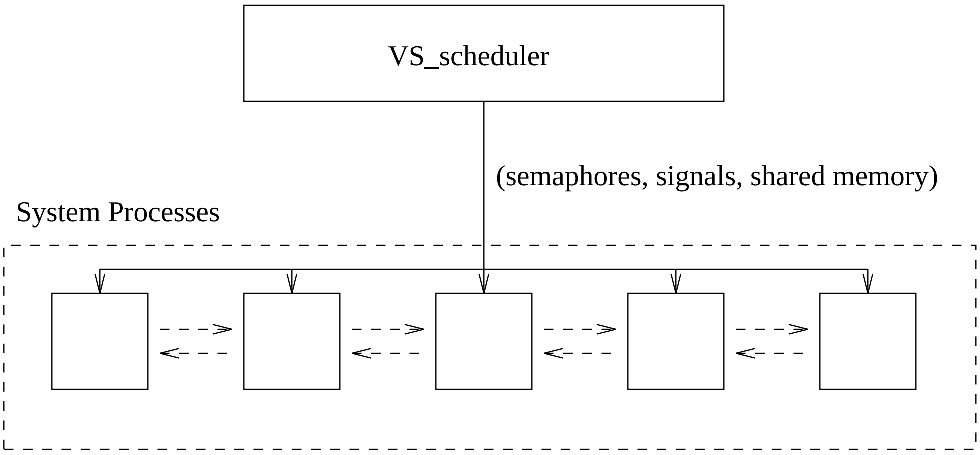

Figure 2: Overall Architecture of VeriSoft (State-Space Search Mode)

be proved [God97] that deadlocks and assertion violations can be detected by exploring only the global states of a concurrent system that satisfies the above assumptions.

VeriSoft can systematically explore the state space of a concurrent system as defined above. In a nutshell, systematic state-space exploration is performed by controlling and observing the execution of all the visible operations of all the concurrent processes of the system. Every process of the concurrent system to be analyzed is mapped to a UNIX process. The execution of the system processes is controlled by an external process, called the *scheduler* (see Figure 2). This process observes the visible operations performed by processes inside the system, and can suspend their execution. By resuming the execution of (the next visible operation of) one selected system process in a global state, the scheduler can explore one transition between two global states in the state space of the concurrent system. By reinitializing the system, the scheduler can explore alternative paths in the state space. Note that there are exactly two sources of nondeterminism in the concurrent systems we consider here, namely concurrency and VS_toss operations, and that the VeriSoft scheduler has complete control over both. Therefore, VeriSoft is always able to completely reproduce any scenario leading to an error found during a state-space search.

Since states of programs written in programming languages can be very complex (because of pointers, dynamic memory allocation, large data structures of various shapes, recursion, etc.), the VeriSoft scheduler does not attempt to compute any representation for the reachable states of the system being analyzed, and hence performs a systematic state-space exploration without storing any intermediate states in memory. It is shown in [God97] that the key to make this approach tractable is to use a new search algorithm built upon existing state-space pruning techniques known as partial-order methods [God96]. For finite acyclic state spaces, this search algorithm is guaranteed to terminate and can be used for detecting deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results. In practice, VeriSoft can be used for systematically and efficiently testing the correctness of any concurrent system, whether or not its state space is acyclic. Indeed, it can always guarantee, from a given initial state, complete coverage of the state space up to some depth.

3 The Heart-Beat Monitor

3.1 Background

The 4ESS switch is a feature-rich, high-capacity toll and tandem switch used by both local and interexchange carriers [ACFM94]. It provides an efficient platform for the routing of telephone calls through a high capacity digital switching fabric. Three elements make up the 4ESS switch system architecture (see Figure 3): the switching processor platform, the switching fabrics platform, and the customer interface platform. The switching processor platform controls the system, ensuring that calls are routed through the switching fabric from telephone user to telephone user. The customer interface platform provides for customer interaction features within the switch.

The primary processor within the switch is the 1B processor. This processor is a special purpose Lucent product tailored to the needs of the 4ESS switch [DMM+95], and has a number of special-purpose functions that facilitate efficient telephony applications. The Common Network Interface (CNI) ring is the switch's interface to common channel signaling links, such as Common Channel Signaling 7 (CCS7) or Q.931 Integrated Services Digital Network (ISDN). A 3B20 processor [S+83] is used within the 4ESS to support the processor platform's disk backup and to connect the 1B processor with the CNI ring; the 3B20 processor is another Lucent Technologies processor product that has found use in several Lucent switching systems such as the 4ESS and 5ESS switches. The CNI ring is a token ring with a number of specialized signaling processors. One node on the CNI ring is the Direct Link Node (DLN): it provides a high speed connection between the 3B20 and the CNI ring,

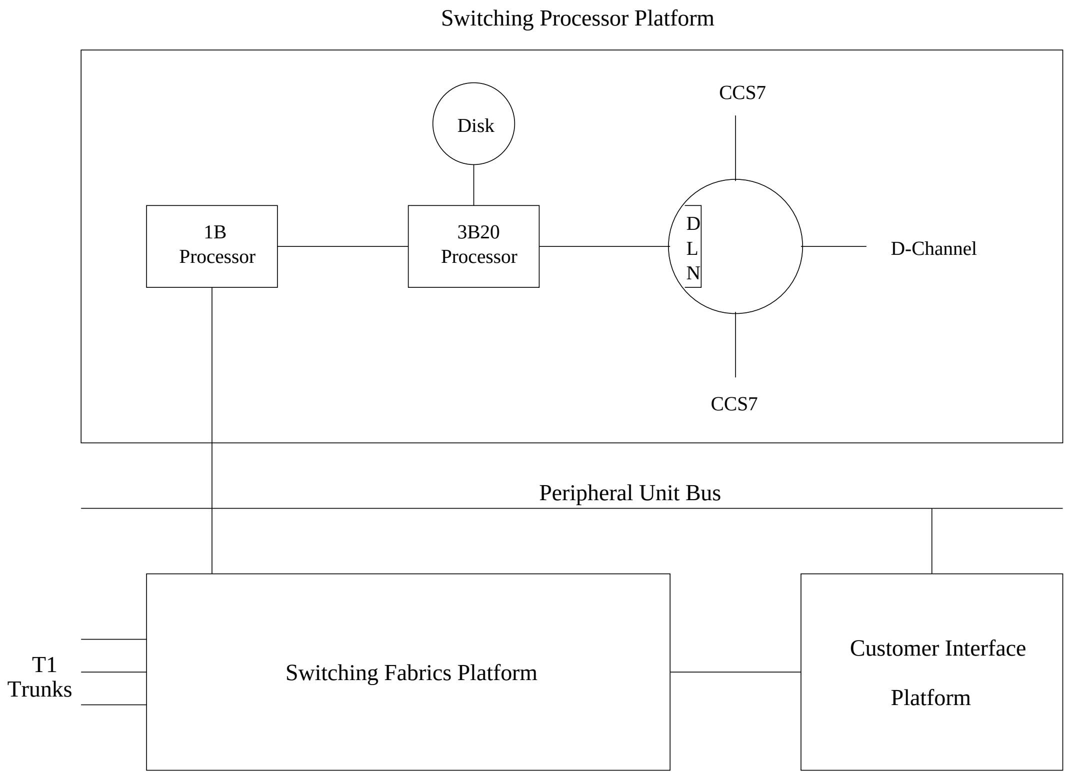

Figure 3: 4ESS Switch Architecture

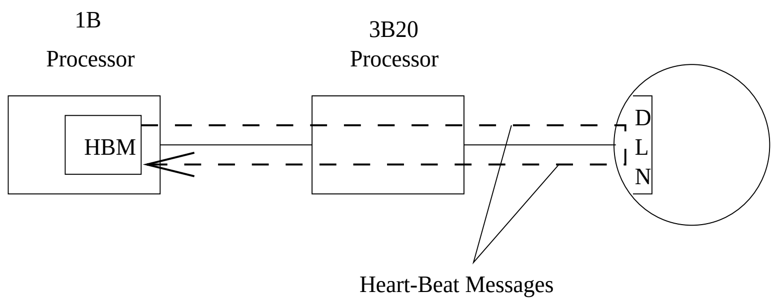

Figure 4: 4ESS Heart-Beat Monitor

and is a member of the Motorola 680x0 processor family.

Since the 1B processor controls call processing, a communications path between the 1B, 3B20 and DLN processors is required to pass messages received from the common channel signaling network. To ensure that these processors are all communicating, the End to End Integrity pattern is used. This pattern describes the need to check for reliable communications at the ends of the path, rather than trusting the individual components along the path to indicate that they are functioning reliably. In the present case, a message (or "heart-beat") is sent from the 1B through the 3B20 to the DLN for return to the 1B to ensure that the entire data path is working (see Figure 4). If there are problems at any point along the way (i.e., in the 1B, the 3B20, the DLN or any of the communication paths) the message will not be reliably returned, indicating the need for some fault recovery action. The Heart-Beat Monitor (HBM) software – which executes on the 1B processor – is responsible for measuring the propagation delays of messages between the 1B and the DLN. Upon receipt of the returned heart-beat messages, the 1B processor determines the elapsed time required by the message to traverse this entire two-way path. If the sequence of message delays become unacceptably high, the HBM software causes signaling messages to be directed away from this path in favor of alternative signaling paths not requiring this chain of processors. This cessation, termed *resource suspension* in this article, is not a desirable state for 4ESS switch operation: it isolates a high volume signaling interface, requiring that alternate, lower volume interfaces be used instead.

The 4ESS switch project has the goal to make the switch as reliable as possible, and hence, the hardware and software components of the switch are continually improved. The HBM is not immune to this improvement process: it was redesigned several years ago to make it less sensitive to transient increases in the propagation delays of heart-beat messages, and is once again being examined in order to improve its functioning. The overall goal is to make the HBM software more robust by eliminating inappropriate invocations of resource suspension.

The first step in this redesign process was to understand thoroughly the behavior of the current version of the HBM software. Since the original developers were no longer available, the current development team had to look at the existing documentation and the actual source code to obtain this understanding. This existing documentation consisted of English language descriptions, illustrative scenarios and hand-generated state-machine tables.

The complexity of the software renders it difficult to model and analyze by hand, and hence a collaboration was begun between research and development to apply automatic verification techniques to aid in understanding the application.

The following sections provide more details about the HBM software and describe our analysis of the software using VeriSoft.

3.2 Details of the Software

The HBM software runs on the 1B processor. Its main procedure is executed at (approximately) fixed intervals of time, called the *scheduling interval* of the HBM. This is also the period of time between two successive heart-beat messages sent by the HBM to the DLN. As shown in Figure 4, these messages are sent via the 3B20 processor, which simply passes them along to the correct destination.

The only constraints that can be assumed about the DLN is that heart-beat messages sent by the 1B processor will not be re-sent in a different order by the DLN. However, there can be arbitrary delays in the re-sending of messages, and messages may be lost. Thus, every heart-beat message sent by the 1B processor must contain some information that uniquely identifies it, and the HBM software must keep track of the time that it was sent.

When a message is received from the DLN, the HBM software calculates the *propagation delay* of the message – namely, the elapsed time between the send-time and the receive-time of the message. In principle, the number of outstanding messages (sent but not yet received) can be unbounded. However, memory and performance considerations of the switch imply that only a small number of outstanding messages can be tracked by the HBM. Similar concerns apply to the count that HBM keeps of messages that have either arrived late or have been deemed as lost.

The HBM software thus keeps a small fixed-size array of messages sent to the DLN. When a message is sent, it is marked with its array index and with a time-stamp indicating the time it was sent. When a message is received from the DLN, a function is (virtually) instantaneously called in the 1B processor to mark the receive-time of the message.

Messages fall into four categories depending on their propagation delay:

- A message is considered to be *on time* if its propagation delay is less than some constant d_{ontime}, which itself is less than the scheduling period of the HBM. Thus, an on-time message will be received before the next invocation of the main HBM procedure.
- A message is considered to be *slightly late* if its propagation delay is greater than the above constant d_{ontime}, but less than the the scheduling period of the HBM. Thus, a slightly late message will also be received before the next invocation of the main HBM procedure.
- A message is considered to be *late/lost* if its propagation delay is greater than the above constant d_{ontime}. Hence, slightly late messages are also considered to be late/lost.
- A message is considered to be *stale* if its propagation delay is greater than k times the scheduling period of the HBM, where k is the size of the message array. Stale messages are deemed as lost and are discarded.

In order to avoid high sensitivity to propagation delays of individual messages, the HBM software is structured in three stages. The first stage indicates that resource suspension has not been triggered in the recent past, the second stage serves to dampen the sensitivity of the HBM to individual delays for a period of time, and the third stage is intentionally sensitive to all delays. The only legal state changes are from the first stage to the second, then to the third, and then back to the first. Changes between stages are triggered by sequences of heart-beat message delays and the passage of time measured in multiples of scheduling periods. Resource suspension can be triggered only in the third stage.

When the main procedure of the HBM software is entered, the message array is searched in a fixed-order for the first entry in which the receive-time is recorded. This indicates that a (non-stale) message with this index was newly received; this message is then processed as described below. This cycle continues until all newly received messages have been processed. Figure 5 gives a simplified pseudo-code description of the message processing algorithm implemented in the actual software.

```
if message arrives on-time
then
    if count > 0
    then count:=count-1
else
    if count ≤ 1
    then count:=count+1;

    if count=2
    then
        if currently in stage 1
        then immediately enter stage 2
        else
            if currently in stage 2 and have spent
              more than d_stage2 time in stage 2
            then immediately enter stage 3;

    if count ≤ 2 and not currently in stage 2
    then count:=count+1;
```

Figure 5: Algorithm for Processing Messages

We make a few observations about the algorithm. In stage 3, the counter is incremented by 2 for a late/lost message, up to a maximum value of 3. However, the counter is decremented only by 1 for a message that arrives on time. This reflects the use of the "leaky bucket counter" pattern (see [ACG+96]) in the design of the HBM. This is intended to make the HBM less sensitive to transient late messages, while guaranteeing that if a certain number of messages arrive late – even interspersed with on-time messages – resource suspension will be triggered. A leaky bucket counter is also used in stage 1, but its maximum value cannot exceed 2 in this stage. The behavior in the second stage is more complicated. Namely, the leaky bucket counter pattern is used only under certain conditions; this provides another dampening effect on the sensitivity to individual delays.

After all of the newly received messages have been processed, the HBM gets ready to send its next heart-beat message, whose index is determined in a round-robin fashion from the message array. If the previously sent message with that index has not yet been received, it is declared to be stale and the late/lost message counter is updated as described in Figure 5 (according to the case in which the message is not on time).

The HBM then checks whether the late/lost message counter is set to 3, its maximum possible value. If this is the case and the HBM is currently in Stage 3, resource suspension is triggered.

Finally, a new heart-beat message is sent to the DLN. The index of the message is the next index above, and the send-time is the current time. The main procedure then exits.

4 Getting Started

In order to analyze the HBM application using VeriSoft, the HBM code has to be *executable*, and an executable representation of the environment in which the HBM operates has to be provided in order to close the system.

The actual HBM software is written in a proprietary assembly language and runs only on the 1B processor. As part of a previous re-engineering effort of applications developed in this language, a compiler from this assembly language to the C programming language had been developed. The compiler-generated C translation of the original HBM assembly code was used for the experiments reported in this paper. Precisely, the output of the compilation is a single C procedure. To obtain an executable program, a simple "wrapper" program that periodically calls this procedure was used.

As mentioned above, the HBM application is just one of the many tasks executed on only one of the processors that can be found in a 4ESS switch. A complete representation of such a complex environment would be far too detailed for an analysis focused only on the HBM code, and was not available anyhow. Therefore, a simplified executable representation of this environment was used in order to simulate its visible behavior.

Our analysis used the following structure of the closed system formed by the HBM and its environment. The DLN is simulated by a separate process implemented in the C programming language, while the transmission medium between the 1B processor and the DLN is modeled by two bounded FIFO message queues, one queue for each direction. Upon receipt of a heart-beat message, the code for the DLN nondeterministically decides (with a call to VS_toss) either to lose this message or to retransmit it with a propagation delay nondeterministically chosen among the relevant categories presented in the previous section. The code for the DLN is designed in such a way that the send/receive-time associated with messages sent/received by the HBM are consistent. For instance, two time-stamps associated with two consecutive messages must be increasing. Also, the receive-time of a message must always be greater than its send-time.

PROPERTY	ANALYSIS
1. If no heart-beat messages ever return to the 1B processor, then resource suspension is triggered for the first time after a_1 intervals.	true
2. If the DLN resends every message *slightly late*, then resource suspension is triggered for the first time after $a_2 (< a_1)$ intervals.	true
3. What is the minimum number of intervals needed to trigger resource suspension?	a_2
4. What is the minimum number of *late/lost* messages needed to trigger resource suspension?	b
5. If messages strictly alternate between being *slightly late* and *on time*, then resource suspension will never be triggered.	false!
6. Stage 2 is always exited exactly c intervals after it is entered.	false!
7. Triggering of resource suspension by the HBM is independent of the system initialization time.	false!

Figure 6: Properties Considered in Our Analysis

To give the reader an idea of the complexity of the system being analyzed, the closed concurrent system composed of the HBM code, its wrapper and the code for the DLN is implemented by several hundred non-commentary source lines (NCSL) of C code. Although the size of this application is very small compared to the total size of all the software needed to control a telephone switch (typically millions of lines of code), it is nevertheless far too complex to be thoroughly and reliably analyzed by manual code inspection (as will become clear in the next section).

Various versions of the environment were also used to test properties of the HBM under specific constraints. For instance, counters can be used to limit the number of late/lost messages. By progressively increasing the maximum value of such counters, VeriSoft can be used to show properties such that "at least b late messages are necessary to force the HBM to trigger resource suspension". Properties that were checked are presented in the next section.

5 Results of Analysis

The starting point of our analysis was a document summarizing the findings of a team of developers who undertook a previous reverse-engineering effort to better understand the HBM software. The document consists of English descriptions, illustrative scenarios, and state machine tables formally specified by hand. In the course of our analysis, we verified and extended some properties given in the document, disproved other properties given in the document, and identified some quite unexpected behaviors of the software.

The properties considered in our analysis are summarized in Figure 6. The assumptions on the DLN and the transmission medium specified earlier are implicit in the properties given in Figure 6 and throughout the following discussion. Figure 6 uses the definitions from the previous section. We note that a_1, a_2, b are small integer constants such that $a_2 < a_1 < 20$, $b < 10$, and $c < 10$. (We cannot reveal the actual values of these constants because of proprietary considerations.)

To analyze these properties using VeriSoft, we modified the HBM code so that an assertion is violated if and only if resource suspension is triggered. Namely, we added a "VS_assert(0)" statement to the (unique) point in the HBM code where resource suspension can be triggered. We then modified the environment from the "Getting Started" section in accordance with the property under analysis, and used VeriSoft to automatically search for scenarios in which the assertion is violated.

Properties 1 and 2 were described in some detail in the document, and hence were the starting point of our analysis. To verify Property 1, we implemented an environment that did not re-send any messages back to the HBM. We then used VeriSoft to automatically analyze the closed system. Since these environment implementations did not contain any nondeterminism, there was a unique minimal scenario demonstrating each property. It was then easy to show that resource suspension is indeed triggered for the first time after a_1 intervals. The verification of Property 2 was analogous.

A natural question that arises in this context is whether there are any possible behaviors of a (legal) DLN and transmission medium under which resource suspension is triggered in fewer than a_2 intervals. In particular, what is the minimum number of scheduling intervals needed to trigger resource suspension? This corresponds to Property 3.

In order to address this question, we used the full environment, described in the "Getting Started" section. We modified the closed system – consisting of the combination of the HBM code and this environment – to terminate after $a_2 - 1$ intervals. VeriSoft systematically searched the state space and discovered that resource suspension was not triggered prior to termination. Together with Property 2, this implies that a minimum of a_2 intervals are needed in order to trigger resource suspension, resolving Property 3. Note that the state of the system at any point in a given scenario can a priori depend on the entire scenario up to that point. Furthermore, this closed system can exhibit at least a few hundred thousand distinct scenarios of length a_2. Hence, it would not have been possible to verify this property without the use of a systematic state-space exploration tool. On a dual processor UltraSparc workstation with 128 MB of RAM, the VeriSoft analysis required about 8 hours, and explored more than 3 million global states of the system. (Since VeriSoft does not store states in memory, run-time rather than memory is always the main limiting factor.)

The next question that arose was about the minimality of messages, rather than intervals. In particular, what is the minimum number of late/lost messages that is needed to trigger resource suspension? This corresponds to Property 4. In order to address this question, we modified our full environment to lose or delay at most a given number of messages. In our first iteration, we specified that at most one message could be late/lost, while all other messages had to be on time. We automatically analyzed the state space and discovered that resource suspension was not triggered up to 10 intervals. (Note that since the state space is infinite, it is impossible to explore it exhaustively.) We then iterated this process, successively incrementing by one the maximum number of late/lost messages. We found that a minimum of b late/lost messages are needed in order to trigger resource suspension within 10 intervals.

We then proceeded to analyze other properties implicit in the reverse-engineering document. In particular, the state tables implied that if messages strictly alternate between being slightly late and on time, then resource suspension will never be triggered. This corresponds to Property 5. We implemented an environment that exhibited exactly this behavior, and analyzed it using VeriSoft. To our surprise (both in the research group and development group), VeriSoft immediately generated a scenario in which resource suspension was triggered! This scenario showed that Property 5 was in fact false. We inspected the scenario through the simulation capabilities of VeriSoft and discovered that the "leaky bucket counter" design of Stages 1 and 3 was not reflected in the state tables.

Proceeding to Property 6, we again used the full environment described in the "Getting Started" section, and augmented the closed system with two boolean variables, *entered* and *exited*, indicating information about Stage 2. In particular, both variables are initialized to false; *entered* is set to true upon entry into Stage 2, and *exited* is set to true upon exit from Stage 2. We then modified the code to indicate an assertion violation when *entered* is true but *exited* is false for a duration of $c+1$ intervals, where c is the fixed time to be spent in stage 2 divided by the scheduling period of the HBM. To our even greater surprise, VeriSoft generated after a few minutes of search a scenario in which this assertion was violated! This violated Property 6, which was explicitly stated in the document and which was a fundamental assumption of the development group: namely, that Stage 2 is always exited exactly c intervals after entry.

At this point, a decision was made by the development group to revise the document based on our findings, and to notify affected parties about the errors discovered in the document.

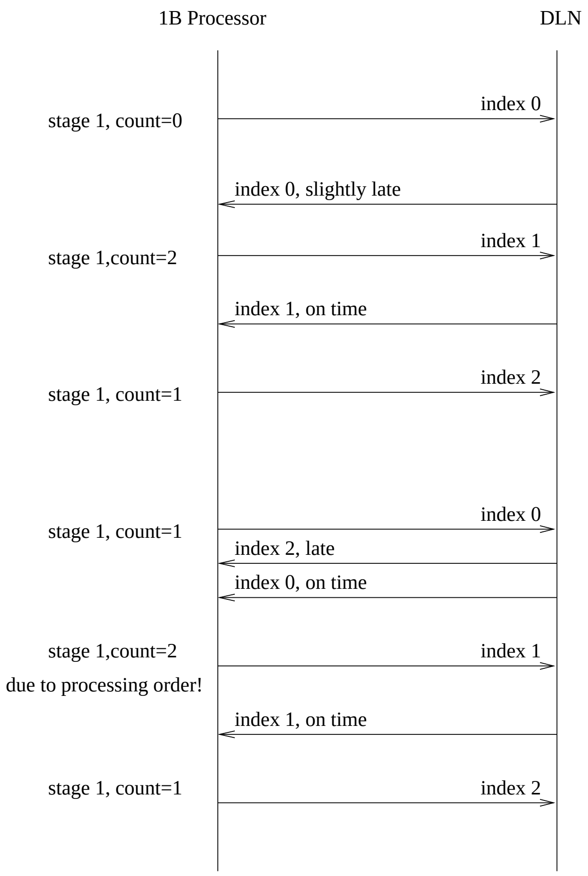

Figure 7: Scenario Related to Property 7

We continued our experiments to see whether any other unexpected behaviors could be revealed. Upon closer inspection of the code through the simulation facilities of VeriSoft, we were rather surprised to find that the double-increment and decrement operations (cf. Figure 5) are not commutative; if the value of the counter is zero or one at the start of an interval, the order in which the messages are processed from the message array can affect the behavior of the HBM during that interval. Furthermore, messages in the message array are always processed in the same fixed order. It occurred to us that this fixed processing order could interact with the non-commutativity of messages in an "irregular" fashion. Using the VeriSoft simulator, we constructed a scenario, depicted in Figure 7, which has the following properties:

- The scenario is executed starting from the initial state of the HBM; at the end of the execution, the HBM is still in Stage 1.
- Suppose a single on-time message is added at the beginning of the scenario. This new scenario is executed starting from the initial state of the HBM; at the end of this execution, the HBM is in Stage 2.

The scenario of Figure 7 can be extended in such a way that the presence or absence of a single additional on-time message at the beginning of the scenario makes the switch trigger resource suspension or not. The existence of this scenario shows that the behavior of the HBM can be dependent on the exact number of intervals during which the system has been running, and hence on the system initialization time! In other words, two switches using this HBM code but initialized at different times may react differently to the same sequence of events, and therefore, their behavior can sometimes be very hard to predict. This pair of scenarios disproves Property 7.

6 Conclusions and Comparison With Other Work

We have presented an analysis of the Heart-Beat Monitor of a telephone switch using VeriSoft, a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary (e.g., C or C++) code. Our analysis of the HBM discovered flaws both in the existing documentation and in the software itself. Based on these findings, we have recently modified the code in several ways to make it more robust and predictable. We then used VeriSoft to systematically test that the desired properties were indeed satisfied by the modified code. Traditionally, the development team has been necessarily cautious in performing any changes in the HBM software, since it can potentially impact the routing of millions of telephone calls per day. However, the confidence gained by our systematic analysis has led the development team to decide to incorporate our modifications into the 4ESS switching software.

With the help of VeriSoft, our analysis revealed HBM behavior that is virtually impossible to detect or test in a traditional lab-testing environment, because of the lack of controllability and observability inherent in such environments. Moreover, running a single test in these environments – which involve actual switching hardware and complex initialization procedures – is much more expensive and clumsy than running thousands of tests, automatically generated, executed and evaluated by VeriSoft, on a standard UNIX workstation. Also, since VeriSoft has complete control over nondeterminism, it can systematically search the state space of a system, and is able to completely reproduce any scenario leading to an error found during the search.

The reason why systematic state-space exploration techniques are increasingly being used is precisely because they can detect and reproduce errors that would be very hard to detect and reproduce otherwise. By extending the scope of these techniques from modeling languages to general-purpose programming languages, VeriSoft eliminates one major obstacle to a wider use of these techniques, namely the need to build a model of the application to be analyzed. The elimination of this time-consuming and error-prone task (plus the non-negligible effort needed to become familiar with a modeling language) makes systematic state-space exploration much more attractive and economically feasible for applications developed in an industrial environment, where systems are always developed under time pressure.

Besides reducing the up-front cost of using systematic state-space exploration, another advantage of VeriSoft is that it exercises the actual code of the concurrent reactive software under analysis. Since most of the time during an analysis is typically spent to examine (user-defined or automatically-generated) scenarios with the interactive simulator, the user's knowledge of the existing code can strongly facilitate the examination of these scenarios. If the code is unknown to the user, VeriSoft can be used to discover the precise dynamic behavior of the application: VeriSoft is WYSIWYG (What You See/Simulate Is What You Get).

This feature of VeriSoft supports new applications for systematic state-space exploration techniques, such as reverse engineering. Indeed, the state space of the actual system contains much information that can be used to better understand how the code is being exercised and how the different processes behave and interact with each other [BG97]. After all, most development efforts are typically spent in studying and modifying existing code. Also, VeriSoft can be useful for regression testing since properties that hold on a previous version of a product can be tested against new versions of the software when modifications are performed.

On the negative side, since VeriSoft does not store any states in memory, it cannot detect cycles in the state space being explored, and hence is restricted to checking safety properties [AS87]. For the same reason, the termination of the search is not guaranteed when the state space contains cycles. (Obviously, even a theoretically-terminating finite-state search might fail to terminate due to the excessive resources that it requires.) This is often not too troublesome in practice since the main goal is to be able to systematically and efficiently search a meaningful portion of the state space within a reasonable amount of time, in order to detect unexpected behaviors of the system. For the application considered here, VeriSoft proved to be a powerful and efficient tool for this purpose.

Note that the size of the state space depends on the nondeterminism in the system, and hence often depends critically on the representation of the (typically nondeterministic) environment of the application being analyzed. Therefore, such an executable representation of the environment has to be developed with care.

Systematic state-space exploration is complementary to other approaches to concurrent reactive program testing and analysis. For instance, *static analysis* techniques (e.g., [CC77, MJ81, ASU86]) automatically extract information about the dynamic behavior of a sequential program by examining its text. Variants of these techniques have also been proposed for the analysis of programs written in concurrent programming languages such as Ada (e.g., [Tay83, LC91, MR93, Cor96]). For specific classes of concurrent programs, these abstraction techniques can produce a "conservative" model of the system that preserves basic information about the communication patterns that can take place in the system. Analyzing such a model using standard model-checking techniques can then prove the absence of certain types of errors in the system. In contrast, our approach is based on the *dynamic* observation of the "actual" processes of the concurrent system. This makes possible a much closer examination of the behaviors of the system, and the detection of a wider range of errors. Moreover, we do not rely on any specific assumption about the static structure of the programs used to represent the behavior of processes, which can actually be written in any language. Interesting future work is to combine the strengths of both the static and dynamic approaches.

VeriSoft also differs from specification-based testing frameworks for reactive programs (e.g., [DY94, Ric94, CRS96, JPP+97]). These techniques compare the input/output behavior of an open reactive program with respect to a high-level specification of its visible behavior. In contrast, VeriSoft was designed to check properties of closed systems composed of multiple processes. It neither enables nor requires the user to provide a precise specification of the input/output behavior of the system to be analyzed. In the case of an open reactive system, it makes it possible to represent the environment of the open system by other processes, and then to check "global" properties of the joint behavior of these processes, in the style of what is usually done with model checking.

Another related and complementary area of research concerns the design of simulators and debuggers for distributed and parallel programs (e.g., [CMN91]). These tools are used to monitor the execution of concurrent processes running in their actual environment. In contrast, VeriSoft has complete control over nondeterminism in order to be able to systematically search the state space of the system for coordination problems. Therefore, it does not preserve quantitative properties (related to timing, performance, etc.) of the whole concurrent system.

Acknowledgments

We thank Lind Weidlich for the use of his compiler in translating the original HBM code into C. An extended version of this paper appeared in [GHJ98]. More information on VeriSoft can be found internally (Lucent intranet only) at `http://www-spr.research.bell-labs.com/~god/verisoft/` and externally at `http://www.bell-labs.com/~god`.

References

[ACFM94] T. W. Anderson, P. D. Carestia, J. H. Foster, and M. N. Meyers. The evolution of the 4ess(tm) switch. *AT&T Technical Journal*, 73(6):93–100, November/December 1994.

[ACG+96] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. Fault-Tolerant Telecommunication System Patterns. In Vlissides, Coplien, and Kerth, editors, *Pattern Languages of Program Design – 2*, pages 549–562. Addison-Wesley, 1996.

[AS87] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BG96] B. Boigelot and P. Godefroid. Model checking in practice: An analysis of the ACCESS.bus protocol using SPIN. In *Proceedings of Formal Methods Europe'96*, volume 1051 of *Lecture Notes in Computer Science*, pages 465–478, Oxford, March 1996. Springer-Verlag.

[BG97] B. Boigelot and P. Godefroid. Automatic Synthesis of Specifications from the Dynamic Observation of Reactive Programs. In *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 321–333, Twente, April 1997. Springer-Verlag.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, January 1977.

[CGH+93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and Their Apllications.* North-Holland, 1993.

[CMN91] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, pages 491–530, October 1991.

[Cor96] J. C. Corbett. Constructing abstract models of concurrent real-time software. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 250–260, San Diego, January 1996.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 1(15):36–72, 1993.

[CRS96] J. Chang, D. Richardson, and S. Sankar. Structural Specification-based Testing with ADL. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 62–70, San Diego, January 1996.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, Cambridge, MA, October 1992. IEEE Computer Society.

[DMM+95] D. C. Dowden, R. E. McLear, A. H. Miller, S. D. Schlough, D. A. Welch, R. A. Wilson, and S. A.

Zeile. Improving on the best: 'Like a 1A, only better'. *AT&T Technical Journal*, 74(3):28–39, May/June 1995.

[DY94] L.K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. *Software Engineering Notes*, 19(5):140–153, December 1994. Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering.

[FGM+92] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *Proc. of the 14th International Conference on Software Engineering ICSE'14*, Melbourne, Australia, May 1992. ACM.

[FHS95] A. R. Flora-Holmquist and M. Staskauskas. Formal validation of virtual finite state machines. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, pages 122–129, Boca Raton, April 1995.

[GHJ98] P. Godefroid, R. S. Hanmer, and L. J. Jagadeesan. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft. In *Proceedings of ACM SIGSOFT ISSTA'98 (International Symposium on Software Testing and Analysis)*, Clearwater Beach, March 1998.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.

[God97] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.

[HK90] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 1990.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[HP89] G. J. Holzmann and J. Patti. Validating sdl specifications: An experiment. In *Proc. 9th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*. North-Holland, 1989.

[JPP+97] L. Jagadeesan, A. Porter, C. Puchol, J.C. Ramming, and L. Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the 19th IEEE International Conference on Software Engineering*, 1997.

[LC91] D. L. Long and L. A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of ACM Symposium on Testing, Analysis, and verification (TAV4)*, pages 21–35, Vancouver, October 1991.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MJ81] S.S. Muchnick and N.D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[MR93] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel programming*, pages 129–138, San Diego, May 1993.

[Ric94] D.J. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, August 1994.

[Rud92] H. Rudin. Protocol development success stories: Part I. In *Proc. 12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, June 1992. North-Holland.

[S+83] J. M. Scanlon et al. The 3B20D processor. *Bell System Technical Journal*, 62(1), 1983.

[Tay83] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, pages 362–376, May 1983.