Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft

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++. The state space of a concurrent system is a directed graph that represents the combined behavior of all concurrent components in the system. By exploring its state space, VeriSoft can automatically detect coordination problems between the processes of a concurrent system.

We report in this paper our analysis with VeriSoft of the "Heart-Beat Monitor" (HBM), a telephone switching application developed at Lucent Technologies. The HBM of a telephone 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.

We discuss the steps of our analysis of the HBM using VeriSoft. Because no modeling of the HBM code is necessary with this tool, the total elapsed time before being able to run the first tests was on the order of a few hours, instead of several days or weeks that would have been needed for the (error-prone) modeling phase required with traditional model checkers or theorem provers.

We then present 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 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 the redesign of the HBM software in the next commercial release of the switching software.

1 Introduction

Systematic state-space exploration, as such or elaborated into temporal-logic model-checking (e.g., [CES86, LP85, QS81, VW86]), is attracting growing attention for checking the correctness of concurrent reactive systems. In the case of a software system, 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]).

Recently [God97], it has been shown how the scope of 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 full-fledged general-purpose programming languages such as C or C++. *VeriSoft* is a tool for systematically and efficiently exploring the state spaces of such systems. It controls and observes the execution of all the concurrent processes of the system, and can reinitialize their executions. VeriSoft can automatically detect coordination problems (deadlocks, divergences, ...) between the concurrent processes of a system, and check for assertion violations. 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.), VeriSoft 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.

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 "Heart-Beat Monitor" (HBM), an application that is part of the software controlling telephone switches developed by

Lucent Technologies. The HBM of a telephone 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. Since no modeling of the HBM code is necessary with our tool, the total elapsed time before being able to run the first tests was on the order of a few hours, instead of several days or weeks that would have been needed for the (error-prone) modeling phase required with traditional model checkers 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 is 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. These results are being used as the basis for the redesign of the HBM software in the next commercial release of the switching software.

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. The paper ends with a comparison of this approach with other approaches.

2 A Quick Introduction to VeriSoft

VeriSoft [God97] is a tool for systematically exploring the state space of a concurrent system composed of a finite set $\mathcal{P}$ of *processes* and a finite set of *communication objects*. Each process $P_i \in \mathcal{P}$ 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 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. Examples of communication objects are shared variables, semaphores, and FIFO buffers. At any time, at most one operation can be performed on a given communication object (operations on a same communication object are mutually exclusive). Operations on communication objects are called *visible operations*, while other operations are called *invisible*. The execution of an operation is said to be *blocking* if it cannot be completed. We assume that only executions of visible operations may be blocking.

The 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 expected to always eventually attempt to execute a visible operation.[1] This 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. We define a *transition* as a visible operation followed by a finite sequence of invisible operations performed by a single process. A transition whose visible operation is blocking in a global state s is said to be *disabled* in s. Otherwise, the transition is said to be *enabled* in s. A transition t that is enabled in a global state s can be *executed* from s. Once the execution of t from s is completed, the system reaches a global state s', called the *successor* of s by t. The *state space* of the concurrent system is composed of the global states that are reachable from the initial global state s_0, and of the transitions that are possible between these.

A concurrent system as defined here is a closed system: from its initial global state, it can evolve and change its state by executing enabled transitions. 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 visible and nondeterministic: the execution of a transition starting with VS_toss(n) may yield up to $n+1$ different successor states, corresponding to different values returned by VS_toss.

It can be shown [God97] that *deadlocks* and *assertion violations* can be detected by exploring only the global states of a concurrent system as defined in the previous section. Deadlocks are states where the execution of the next operation of every process in the system is blocking. 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 is considered visible. It 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*.

VeriSoft is a tool for systematically exploring the state space of a concurrent system as defined above. In a nutshell, 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*. 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. The scheduler also contains an implementation of a new search algorithm that makes it possible to systematically and efficiently explore the state spaces of such systems without storing any intermediate states in memory. This algorithm is 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. (See [God97] for details.)

When an error of the type listed above is detected during state-space exploration, a scenario leading to the er-

[1] 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".

ror 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. 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.

3 The Heart-Beat Monitor

3.1 Overview

In telephone switches, calls are typically routed through a network of hardware devices, involving a distributed set of processors. In order to ensure the reliability of calls, switches can determine the status of their processors by measuring propagation delays of messages transmitted between them. Longer than expected delays may indicate potential problems; the switch may then temporarily cease to connect new telephone calls over all hardware units connected to the offending processor. This can significantly impact switch performance if the switch normally relies on these units to carry a substantial proportion of telephone calls.

We have analyzed the "Heart-Beat Monitor" (HBM) software of a Lucent switching system. This software is responsible for measuring the propagation delays of messages between two processors A and B. The HBM, running on processor A, periodically sends a "heart-beat" message to processor B. Upon receipt of the heart-beat message, processor B responds by sending an acknowledgment back to processor A. The HBM monitors the delay between the transmission and acknowledgment of messages. If such delays become unacceptable, HBM temporarily ceases the routing of all new telephone calls over processor B – this is termed as *resource suspension* in this paper.

The decision to trigger resource suspension involves some tradeoffs. Calls routed over processor B may not behave reliably in the face of unacceptable propagation delays; on the other hand, resource suspension may cause many new calls to be blocked. Clearly, end-users may become irate in either situation; furthermore, switch operators are required by law to report every extended occurrence of call blocking to the Federal Communications Commission. Hence, the HBM software is carefully engineered to achieve a reasonable balance between switch reliability and switch capacity.

The HBM software of this Lucent switch came under scrutiny a few years ago, since it was discovered in the field that resource suspension was occurring too frequently and resulting in a significant decrease in network capacity. In particular, the software was deemed too sensitive to the delays of individual messages. The software was then modified to track delays between successive messages, and to trigger resource suspension only when delays had been unacceptable for some significant period of time. This modified software has been running for the past several years in over a hundred switches in the field.

Recently, the software came under scrutiny again, this time because of questions concerning unexplained resource suspensions observed in the field. Since the original development team was unavailable, a new team performed reverse-engineering on the actual code. Their findings were summarized in a document consisting of English descriptions, illustrative scenarios, and formal state-machine tables, all produced by hand after inspecting the code. The intent of this document was to serve as a specification of the software, both for internal Lucent use in addressing customer queries and as a basis for future development.

However, the complexity of the software renders it difficult to model and analyze. A research/development collaboration was begun to investigate whether automatic verification techniques based on model-checking would aid in analyzing this 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 processor A. The only constraints that can be assumed about processor B is that heart-beat messages sent by processor A will not be re-sent in a different order by processor B. However, there can be arbitrary delays in the re-sending of messages, and messages may be lost. Thus, every heart-beat message sent by processor A 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 processor B, the HBM software calculates the delay 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 require 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 processor B. When a message is sent, it is marked with its array index and with a time-stamp indicating the time it was sent. The HBM also keeps track of the index of the last message sent.

When a message is received from processor B, a function is (virtually) instantaneously[2] called in processor A to mark the receive-time of the message in the array entry corresponding to the index of the message, provided the send-time of the message matches the send-time recorded in the array entry. Otherwise, the received message is deemed to be *stale* and is discarded.

In order to avoid high sensitivity to delays of individual messages, the HBM software is structured in three stages. At any given time, the HBM is in exactly one of these 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. After spending a fixed period of time in the third stage, the software re-enters the first stage. We emphasize that resource suspension can be triggered only in the third stage.

In order to more precisely describe the application, we use the following definitions and notation. We cannot reveal the actual values of the constants $k, d_{\mathrm{ontime}}, d_{\mathrm{stage2}}, d_{\mathrm{period}}$

[2] Note that this function is not called when the HBM is being executed. Instead, the function is called after the current execution of the HBM is completed, and it takes the running time of the HBM into account in calculating the receive-time of the message.

below because of proprietary considerations. To give an intuition for the HBM behavior, we give approximate ranges for some of these constants.

- k is the size of the message array. We note that k is a small integer constant, less than 10.
- The main procedure of the HBM software is executed (approximately) every d_{period} time units. We call each scheduling of the HBM an *interval.* d_{period} is also the (approximately) fixed period between successive heart-beat messages sent by the HBM.
- The *propagation delay* of a message that is sent but never received by the HBM is ∞. Otherwise, the propagation delay is t_2-t_1, where t_1 is the time the message was sent by the HBM, and t_2 is the time it was received by the HBM.
- d_{ontime} is a non-zero integer constant strictly less than d_{period}.
- A message is considered to be *on time* iff its propagation delay is less than or equal to d_{ontime}.
- A message is considered to be *slightly late* iff its propagation delay is strictly between d_{ontime} and d_{period}.
- A message is considered to be *late/lost* iff its propagation delay is strictly greater than d_{ontime}. (Hence, slightly late messages are also considered to be late/lost.)
- A message is considered to be *stale* iff its propagation delay is less than ∞, but strictly greater than $k \times d_{\text{period}}$.
- d_{stage2} is the minimum amount of time that the HBM must remain in stage 2 after entering it. We note that $\lceil d_{\text{stage2}}/d_{\text{period}} \rceil < 10$.

The main abstraction used by the developers in reverse-engineering the code concerns the real-time behavior of the HBM. In particular, since the HBM is scheduled at (approximately) fixed intervals of d_{period} time units, the developers abstract the passage of time as a discrete function of these intervals.[3] For example, the HBM must remain in the second stage for a minimum of d_{stage2} time units. The corresponding abstraction used by the developers states that the HBM must remain in the second stage for a minimum of $\lceil d_{\text{stage2}}/d_{\text{period}} \rceil$ intervals.

When the main procedure of the HBM software is entered, it first checks whether the HBM is in the third stage; if it has already spent the required amount of time in this stage, the HBM re-enters the first stage. Next, the message array is searched in a fixed-order (i.e. lowest index to highest index) for the first entry in which the receive-time is recorded. This indicates that a (non-stale) message with this index was newly received (i.e. after the previously scheduled execution of the HBMsoftware); this message is then processed as described in detail below. This cycle continues until all newly received messages have been processed.

The outstanding messages (sent but not received) are then evaluated, the determination of whether to trigger resource suspension is made, a new heart-beat message is sent to processor B, and the procedure exits. These steps are described in detail below.

[3] This abstraction reflects the essentially deterministic scheduling algorithm used in this particular switch, and is used in developing, maintaining and reasoning about many other software systems in this switch.

3.2.1 Processing of Received Messages

A message is processed as described in Figure 1. The HBM keeps a fixed-size counter of the number of late/lost messages; the counter being incremented to its maximum size is an indication that resource suspension should be triggered. If the received message is on time, the late/lost message counter is decremented by 1 if it is currently non-zero. Otherwise, the behavior is essentially dependent on three factors: the value of the late/lost message counter, the current stage of the HBM, and the amount of time it has been in that stage. Figure 1 gives a simplified pseudo-code description of the algorithm implemented in the actual software. The "if-then(-else)" clauses are executed in succession in the case where the message does not arrive on-time.

```
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 1: 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.

3.2.2 Evaluation of Outstanding Messages

After all of the newly received messages have been processed, the HBM gets ready to send its next heart-beat message. Let i be the index of the previously sent message. Then the index j of the next message to be sent is $(i+1) \bmod k$, where k is the size of the message array. (We note that the array is zero-indexed: namely, the lowest index of the array is 0, while the highest index of the array is $k-1$.)

The HBM now looks in array entry j. If the corresponding message has not yet been received – the send-time has been recorded but the receive-time has not – then the message is declared as lost, and the late/lost message counter is updated as described in Figure 1 according to the case in which the message is not on time.[4]

3.2.3 Determination of Resource Suspension

The HBM then checks whether the late/lost message counter is set to 3, its maximum possible value. (This can be the case only if the HBM is currently in Stage 3.) If so, resource suspension is triggered.

3.2.4 Sending of Next Heart-Beat Message

Finally, a new heart-beat message is sent to processor B. The index of the message is the index j above, and the send-time is the current time. The send-time is also recorded in the corresponding array entry. The main procedure then exits.

4 Getting Started

As previously explained, in order to analyze the HBM application using VeriSoft, the HBM code has to be *executable*, and the system has to be *closed*, i.e, an executable representation of the environment in which the HBM operates has to be provided.

The actual HBM software is written in a proprietary assembly language and runs only on a proprietary special-purpose processor for telephone switches. 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 a self-contained 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 telephone 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.

The specification of the environment of the HBM is as follows. Messages sent to processor B may be lost, but may not be reordered. Moreover, the send/receive-time associated to messages sent/received by the HBM have to be consistent. For instance, two time-stamps associated with two consecutive messages must be increasing. Also, the receive-time has to always be greater than the send-time.

The structure of the closed system formed by the HBM and its environment used in our analysis is the following. Processor B is simulated by a separate process implemented in the C programming language, while the transmission medium between processor A and B is modeled by two bounded FIFO message queues "AtoB" and "BtoA", one queue for each direction. The process corresponding to the HBM (process A) periodically sends a heart-beat message to process B via the queue "AtoB", and then waits to receive messages on the queue "BtoA".

The code for processor B starts by waiting for a heart-beat message from the HBM. Once it receives such a message, it nondeterministically decides (with a call to VS_toss) either to lose this message or to store it in an internal queue of size k, where k is the size of the array used in the HBM to store messages. Let n be the number of messages currently stored in this internal queue. If $n = k$, then the oldest message is sent back to HBM. (See below for an explanation of why this is done.) Then, a number m between 0 and n is picked nondeterministically, and the m oldest messages in the queue are sent back to the HBM. Finally, a special message "tick" is sent to the wrapper of the HBM to indicate that the HBM procedure can be called.

Of course, other representations for the environment are also possible. This specific representation was chosen because it made it easy to observe the exchange of messages between the two processors, since sending and receiving messages via message queues are visible operations according to the VeriSoft terminology.

Note that we attempted to reduce nondeterministic choices as far as possible in the representation of the environment of the HBM. For instance, when $n = k$, we force the oldest message to be retransmitted. This optimization is correct because re-transmitting this message at a later time would cause it to be stale since its slot in the k-size array of the HBM would then be taken by a new fresh message with the same index but a more recent send-time.[5] Since stale and lost messages have the same impact on the HBM logic, it suffices to consider only message loss.

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 Processor B 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 lost and/or delayed 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

As mentioned earlier, a team of developers undertook a reverse engineering effort to better understand the HBM software, and a document was produced summarizing their findings. The document consists of English descriptions, illustrative scenarios, and state machine tables formally specified

[4]This justifies discarding stale messages that have index j but an earlier send-time. Namely, the loss of such messages has already been counted in this step.

[5]This property of the HBM was actually tested on a previous less-optimized representation of the environment.

PROPERTY	ANALYSIS
1. If no heart-beat messages ever return to processor A, then resource suspension is triggered for the first time after a_1 intervals.	true
2. If B 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 $\lceil d_{\text{stage2}}/d_{\text{period}} \rceil$ intervals after it is entered.	false!
7. Triggering of resource suspension by the HBM is independent of the system initialization time.	false!

Figure 2: Properties Considered in Our Analysis

by hand. This document was the starting point of our analysis of the HBM software. 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 and irregular behaviors of the software.

The properties considered in our analysis are summarized in Figure 2. The assumptions on processor B and the transmission medium specified earlier are implicit in the properties given in Figure 2 and throughout the following discussion. Figure 2 uses the definitions from the previous section. We note that a_1, a_2, b are small integer constants such that $a_2 < a_1 < 20$ and $b < 10$; we cannot reveal the actual values because of proprietary considerations.

To analyze these properties using VeriSoft, we modified the HBM code so that an assertion is violated iff 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 detect the conditions under which the assertion can be 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) processor B and transmission medium under which resource suspension is triggered in fewer than a_2 intervals. In particular, what is the minimum number of 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. We automatically analyzed the state space using VeriSoft 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 path can a priori depend on the entire path 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 millions global states of the system.[6]

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.[7] 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 document. In particular, the state tables imply 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! We inspected the scenario through the simulation capabilities of

[6] Since VeriSoft does not store states in memory, run-time rather than memory is always the main limiting factor.

[7] Note that since the state space is infinite, it is impossible to explore it exhaustively.

VeriSoft and discovered that the "leaky bucket counter" design of Stages 1 and 3 was not reflected in the state tables. As described in Figure 1, in these stages the software double-increments the count upon the arrival of every (slightly) late message, but only single-decrements the count upon the arrival of every on-time message. The state tables incorrectly specified that the incrementing was by one rather than two, and hence that Stage 2 and Stage 3 were never entered in this environment. In contrast, we discovered that Stage 2 can be entered in the code during the third interval, is exited after $\lceil d_{\text{stage2}}/d_{\text{period}} \rceil$ intervals, and that resource suspension is triggered in Stage 3 because of the double increment. This scenario showed that Property 5 was in fact false.

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 $\lceil d_{\text{stage2}}/d_{\text{period}} \rceil + 1$ intervals. 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 $\lceil d_{\text{stage2}}/d_{\text{period}} \rceil$ intervals after entry. In this VeriSoft-generated scenario, the initial two messages were slightly late; thus, the counter was incremented to two, and Stage 2 was entered. The next two messages were on time; hence, the counter was decremented to zero. All the following messages strictly alternated between being slightly late and being on time. The subsequent behavior of the HBM is best explained via examination of Figure 1. Since Stage 2 does not use the leaky bucket counter pattern, the counter is only incremented by 1 by a late/lost message, and still decremented by 1 by an on-time message. Thus, the counter toggled between zero and one; since it never reached a value of 2, the check was never done for whether d_{stage2} amount of time had expired since entry into Stage 2. Consequently, Stage 2 was not exited. This scenario continues to hold even 100 intervals after entry into Stage 2. Furthermore, it is in contrast to Property 5, in which alternating messages do lead to resource suspension.

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.

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 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. Specifically, the processing order can affect the value of the counter at the end of the interval, and – even more significantly – whether Stage 2 is entered from Stage 1 during the interval. These differences are a consequence of the algorithm sketched in Figure 1.

As mentioned earlier, the messages are always processed in fixed order: from lowest to highest index. It occurred to us that this fixed processing order could interact with the

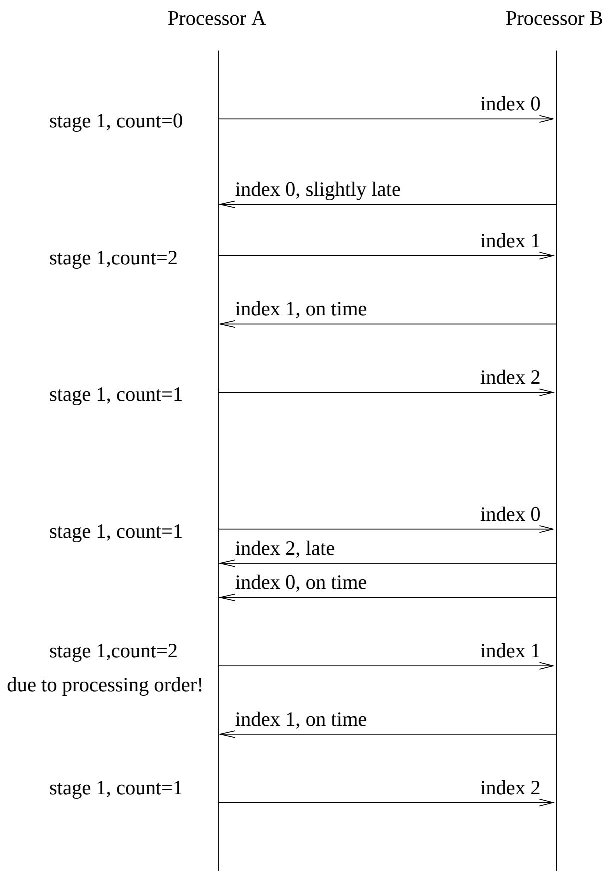

Figure 3: Scenario related to Property 7

non-commutativity of messages in an "irregular" fashion. Recall that 0 is the lowest index and $k-1$ is the highest index. Suppose that the HBM is in Stage 1, the counter value is currently 1, a message with index $k-1$ is more than slightly late, and the next message (i.e. with index 0) is on time. In particular, both of these messages are processed in the same interval. Because of the fixed order of processing, the on-time message will be processed first. We constructed a scenario in which Stage 2 is not entered after these messages are processed. This scenario is heavily dependent on the fixed processing order of messages, and is depicted in Figure 3; for illustrative purposes, we assume $k = 3$. The algorithm in Figure 1 is crucial to understand this scenario. In the fourth interval of this scenario, the counter value is initially 1. When the on-time message with index 0 is processed, the counter becomes 0. After the late message with index 2 is processed, the counter becomes 2, but Stage 2 is not entered. (Note that the opposite order of message processing would cause the HBM to enter Stage 2 after the processing of the late message, and to remain in Stage 2 after the processing of the on-time message.)

Now suppose that we add a single on-time message to the beginning of this scenario. Thus, the HBM stays "clean" for one interval, and then the scenario in Figure 3 is executed (except that the indices of the messages are shifted by 1). In the fifth interval of this new scenario, a message with index 0 is more than slightly late, and the next message (i.e. with index 1) is on time. However, this time the late message is processed first, and the HBM enters Stage 2.

These scenarios can be extended so that the next $\lceil d_{\text{stage2}}/d_{\text{period}}\rceil - 1$ messages arrive on time, and the following two messages arrive late. In the first extended scenario, the last message causes the HBM to enter Stage 2, and resource suspension thus cannot be triggered for a minimum of d_{stage2} time units. In the second extended scenario, however, the last message causes the HBM to enter Stage 3 and trigger resource suspension. Hence, since the only difference between the two scenarios is the existence of an initial on-time message, the behavior of the HBM is dependent on the exact number of intervals 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. 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 quite reluctant to perform any changes in the HBM software since it can potentially impact the routing of millions of telephone calls per day. However, in this case, the confidence gained by our systematic analysis has led the development team to decide to incorporate our modifications into the next commercial release of the 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.[8] This is often not very 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.

[8] Obviously, even a theoretically-terminating finite-state search might fail to terminate due to the excessive resources that it requires.

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.

Ackowledgments

We thank Lind Weidlich for the use of his compiler in translating the original HBM code into C.

References

[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.

[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.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[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.

[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.

[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.

[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.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[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.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[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.

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

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.