Software Model Checking in Practice: An Industrial Case Study

Satish Chandra
Bell Laboratories
Lucent Technologies
schandra@bell-labs.com

Patrice Godefroid
Bell Laboratories
Lucent Technologies
god@bell-labs.com

Christopher Palm
Wireless Network Group
Lucent Technologies
snow72@lucent.com

ABSTRACT

We present an application of software model checking to the analysis of a large industrial software product: Lucent Technologies' CDMA call-processing library. This software is deployed on thousands of base stations in wireless networks world-wide, where it sets up and manages millions of calls to and from mobile devices everyday. Our analysis of this software was carried out using VeriSoft, a tool developed at Bell Laboratories that implements model-checking algorithms for systematically testing concurrent reactive software.

VeriSoft has now been used for over a year for analyzing several releases and versions of the CDMA call-processing software. Although we started this work with a fairly robust version of the software, the application of model checking exposed several problems that had escaped traditional testing. Model checking also helped developers maintain a high degree of confidence in the library as it evolved through its many releases and versions.

To our knowledge, software model checking has rarely been applied to software systems of this scale. In this paper, we describe our experience in applying this technology in an industrial environment.

1. INTRODUCTION

During the last twenty years, tremendous progress in the development of hardware and communication technologies has enabled the emergence of a new class of programs: *reactive programs*. While traditional programs transform an input into an output in order to *compute something*, reactive programs continually interact with their environment in order to *control something*. Nowadays, reactive programs are ubiquitous in our lives: they control telephones, airplanes, ATMs, power plants, pacemakers, etc.

With the rise of reactive programs has grown what has become the method of choice for verifying their correctness: *model checking* [14, 6]. The term "model checking" means to check whether all possible behaviors of a reactive program are "models", in the classical logical sense, of a temporal-logic formula representing a property. Since the 1980's, the study of model checking has been an active area of research, leading to significant new results on temporal logics, automata theory, process algebras and state-space exploration algorithms.

In the early 1990's, model checking evolved from a mostly theoretical discipline in computer science to a practical design-verification framework thanks to the development of tools such as CAESAR [15], COSPAN [21], CWB [7], MURPHI [12], SMV [28] and SPIN [22], among others. These tools can automatically explore the state space of a concurrent/reactive system represented by a program specified in essentially a finite-state *modeling language*. They have been successfully applied to analyze the correctness of a large variety of reactive systems, ranging from circuit designs to communication protocols.

A typical success story in this context would consist of showing that a thorough analysis of a system using model checking had revealed important flaws previously unknown to the designers (for instance, see [5, 3]). The main practical contribution of model checking is thus that it *can expose subtle design errors* that would be very hard to find otherwise. Although model checking is a verification framework, it is closer to *testing* in practice since any verification process is inherently incomplete: only some abstract models or system configurations can be checked against some properties in some environment, and verification results can also be approximate when an exact answer is too expensive to compute.

During the last five years or so, researchers have started investigating one of the most challenging open problems related to model checking: how to apply model checking to analyze reactive *software*. By software, we do not mean models of software systems specified in some finite-state modeling language supported by traditional model-checking tools, even when these models can be compiled to form the core of software implementations as advocated in top-down design methodologies supported by languages such as SDL [23], ESTEREL [2], and VFSM [17]. By software, we mean software written in *programming languages*, such as C, C++ or Java, and of *realistic size*, i.e., possibly hundreds of thousands lines of code. While modeling languages are basically notations for concurrent/extended finite-state machines, programming languages are much more expressive and complex since they support procedures, recursion, dynamic data structures of various shapes and sizes, pointers, etc.

Essentially two approaches to software model checking have been proposed and are still actively being investigated. The first approach [19] consists of adapting model checking into a form of systematic testing that simulates the effect of model checking while being applicable to processes executing arbitrary code: like a traditional model checker explores the state space of a system model defined as the product of its concurrent finite-state components, one can explore the state space of a software application by first defining it as the "product" of its concurrent (Unix-like) processes and then by using a run-time scheduler for driving the entire application through all the states and transitions in its state space defined this way; this approach is developed in the tool VeriSoft. The second approach consists of automatically extracting a model out of a software application by statically analyzing its code and abstracting away details, and then applying traditional model checking to analyze this abstract model; examples of tools that follow this paradigm are Bandera [10], Feaver [24], JavaPathFinder [32] and SLAM [1]. We will discuss and compare both approaches in detail later.

A few applications of software model checking have been reported in the research literature so far. For instance, [20] describes the analysis of a critical component of a telephone switch using VeriSoft, [24] reports the analysis of a network server product using Feaver and [32] discusses the analysis of a remote-agent spacecraft controller and of the kernel of an avionics operating system using JavaPathFinder. It is worth noting that the software analyzed in each case was described by at most a few thousands lines of code. These encouraging applications demonstrate that software model checking is applicable for so-called *unit testing*, where a rather small, well-delineated, yet complex and critical software application is the focus of the analysis.

In this paper, we report our experience applying software model checking to a much larger software application: Lucent Technologies' CDMA call-processing library. This critical software application runs on every CDMA *base station* sold by Lucent Technologies. A base station is a device containing antennas and radio control hardware and software to manage wireless transmissions. A typical wireless network contains thousands of base stations which may be geographically distributed over large areas. The call-processing library that is the focus of the present work is the software that sets up and manages calls to and from mobile devices (such as cellular phones). In addition to setting up a call initially, call-processing software must also support continuity of connection as a mobile device changes location, also called a *hand-off*. In the third-generation wireless networks, which use CDMA technology, call-processing software is further complicated because multiple logical traffic channels must be maintained for each mobile to support high-quality hand-offs. Since these logical channels are a shared resource at a base station, the call-processing software must implement complex dynamic resource allocation algorithms to manage them.

CDMA call processing presents several challenges from a testing perspective. The first obvious challenge is the large number of possible scenarios that a set of mobile calls can go through; traditional testing techniques can only provide limited test coverage since they depend on the number of scenarios that a human tester can create and execute. Second, the call-processing module is not a simple, single-process, stand-alone application: it is embedded in a highly networked environment composed of multiple processes, invoking call-processing functions through multiple interfaces. Third, the size of the whole application, i.e., hundreds of thousands lines of mostly C and C++ code, and the complexity of its architecture makes any kind of manual or automatic "model extraction" via static analysis problematic, and hence unrealistic under the constraints (time pressure and cost) faced by an industrial development organization.

Our solution takes advantage of VeriSoft, a general-purpose "model checker" for systematically testing concurrent reactive software [19]. We have integrated VeriSoft into the existing execution environment for Lucent's wireless product platform and with the testing interface available for the CDMA call-processing software. We then created several nondeterministic C programs representing in a very compact way a multitude of realistic combinations of external events the software should be able to handle. VeriSoft was then used to systematically drive the execution of these nondeterministic programs through all their possible behaviors, thus exercising the call-processing software through millions of scenarios. More than 1,500 runs of VeriSoft over a period of more than a year have been executed for analyzing several releases and versions of the CDMA call-processing software. This thorough testing exposed several previously unknown problems (software bugs) in various components of the CDMA product.

The rest of this paper is organized as follows. In the next section, we briefly recall the main ideas and features of VeriSoft. In Section 3, we present basic notions of wireless networks and the role of the CDMA call-processing library considered here. In Section 4, we describe the steps necessary to carry out the analysis of the call-processing library using VeriSoft. We then discuss in Section 5 the results obtained and impact of this work. In Section 6, we discuss the costs and limitations of our approach, and Section 7 compares it with other approaches to software model checking and automatic test generation. We conclude in Section 8 with some general remarks on technology-transfer and business issues related to software model checking.

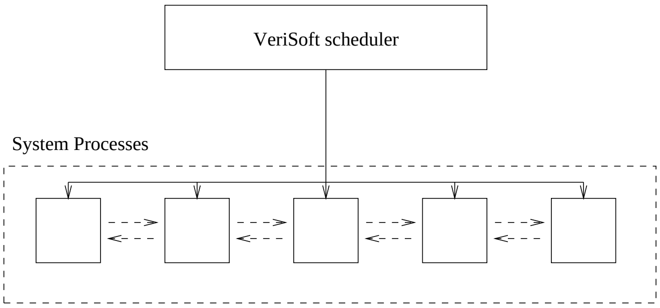

Figure 1: Overall Architecture of VeriSoft (in Automatic State-Space Exploration Mode)

2. MODEL CHECKING SOFTWARE WITH VERISOFT

We briefly recall in this section the main ideas of the framework introduced in [19]. We consider a concurrent system composed of a finite set of *processes* and a finite set of *communication objects*. Each process executes a sequence of *operations* 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 atomic operations on communication objects, such as shared variables, semaphores, and FIFO buffers. Operations on communication objects are called *visible operations*, while other operations are by default called *invisible*. 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 visible operations may be blocking.

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 executing a visible operation. (If a process does not attempt to perform a visible operation within a given amount of time, an error, called divergence, is reported at run time.) This assumption implies that initially, after the creation of all the processes of the system, the system can reach a first and unique global state s_0, called the *initial global state* of the system. A *process transition* is defined as 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.

VeriSoft is a tool for systematically exploring the state space of a concurrent system as defined above. Systematic state-space exploration is performed by controlling and observing the execution of all the visible operations of the concurrent processes of the system. The execution of the system processes is controlled by an external process, called the *scheduler* (see Figure 1). 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. It is thus assumed that there are exactly two sources of nondeterminism in the concurrent systems considered here: concurrency and calls to a specific visible operation named `VS_toss` used to model nondeterminism as described below. When this assumption is satisfied, the VeriSoft scheduler has complete control over nondeterminism, and can thus reproduce any scenario leading to an error found during a state-space search.

As with most systematic state-space exploration tools, VeriSoft requires an executable representation of the environment (test driver) in which the system operates, in order to close the system and make it self-executable. For this purpose, the special visible 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`.

Four main classes of errors can be detected by VeriSoft: *deadlocks*, *livelocks*, *divergences* and *assertion violations*. Deadlocks are states where the execution of the next operation of every process in the system is blocking. A livelock occurs when the execution of the next visible operation of some process is blocking during a sequence of more than a given (user-specified) number of successive states in the state space. Deadlocks and livelocks are notorious problems in concurrent systems, and are extremely difficult to detect through conventional testing methods. A divergence occurs when a process does not attempt to execute a visible operation within a specified (bounded) amount of time. Divergences may be caused by segmentation faults, non-terminating loops, etc. Assertions can be specified anywhere in the application code with the special operation `VS_assert`. This operation can be used in 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.

Since states of programs 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 stor-

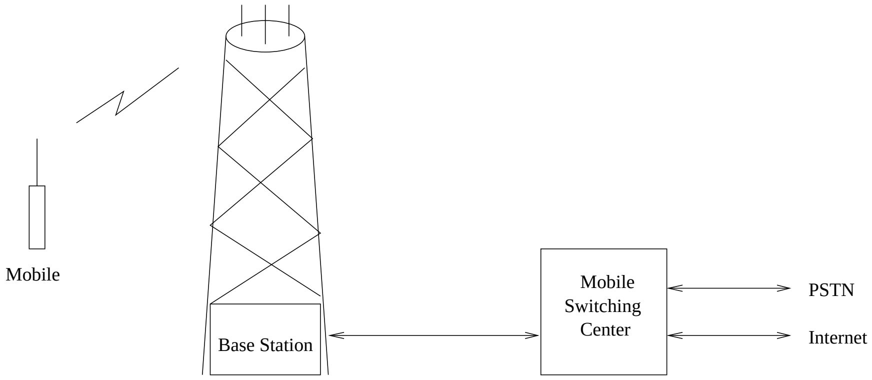

Figure 2: Main Elements of a Wireless Network

ing any intermediate states in memory. It is shown in [19] that the key to make this approach tractable when exploring the state spaces of concurrent systems is to use a new search algorithm built upon existing state-space pruning techniques known as partial-order methods [18]. 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 [19].

In practice, the user has the responsibility of declaring what operations are visible to VeriSoft. Obviously, if all the sources of nondeterminism are not under the control of VeriSoft, completeness of verification results and reproducibility of error traces cannot be guaranteed anymore. Hiding operations and processes to VeriSoft can be done on purpose to be able to analyze very large applications. For instance, an entire communication switch can be be viewed as a huge black-box and multiple concurrent test-drivers controlled by VeriSoft can simulate various sequences of external events occurring at different interfaces of the switch (simulating other switches, user traffic and hardware failures, for example); even though VeriSoft does not control the nondeterminism (if any) inside the black-box itself with this approach, this can still be a very challenging test for the application. This type of approximation is necessary for analyzing applications of the size and complexity of the CDMA call-processing software.

VeriSoft can be run in two modes. In automatic state-space exploration mode, VeriSoft systematically searches the state space of the application for errors. In manual simulation mode, a user can interactively explore specific paths in the state space of the system, such as paths leading to errors found automatically from a previous state-space search; the VeriSoft simulator provides a graphical user interface for easily replaying and examining test scenarios.

3. CDMA CALL-PROCESSING SOFTWARE

A wireless communication network typically contains three main types of network elements: *mobiles* (e.g., cellular phones) which are mobile communication devices; *base stations* (or *cell sites*, or *cells*) which contain antennas and radio control hardware and software to manage the air interface; and *mobile switching centers* which are larger switches handling wireless specific features such as location management (to keep track of where mobiles are located) and also interface with core networks such as the public-switched telephone network (PSTN) and the internet. Figure 2 shows these components. For the purpose of this paper, we will be concerned with testing the part of call-processing software that resides in a base station.

The most critical resource in a wireless system is the radio frequency spectrum. Each carrier in a given geographical area buys exclusive rights to a portion of the radio spectrum from an administrative entity (e.g., the FCC in the US), and must use that portion to support all subscribers in that region. Therefore, it is imperative to use the spectrum as efficiently as possible, while maintain acceptable quality of service to subscribers. In older, so-called “first generation”, analog technology, each carrier divided its portion of spectrum into fixed 30 kHz bands, where each band supported one call. To increase capacity, the second-generation digital systems use one of two techniques, TDMA (time-division multiple access) or CDMA (code-division multiple access). In TDMA, to increase system capacity, each 30 kHz band is shared by multiplexing digitized voice from multiple calls into time slots. In CDMA, however, the entire carrier's spectrum is shared by all calls; the distinction between calls is made by encoding and decoding each call using a separate key. A commonly used key scheme is called *Walsh* codes, which have the property that they are *orthogonal* in vector space sense. To pick out a call coded with Walsh code a from other calls coded with other Walsh codes b, c, etc., a receiver simply multiplies the input with code a, which effectively zeroes out the contribution of other calls. The number of voice channels in the system is determined by the number of orthogonal codes supported by the system, typically 64 for 1.25 MHz spectrum. Today, CDMA technol-

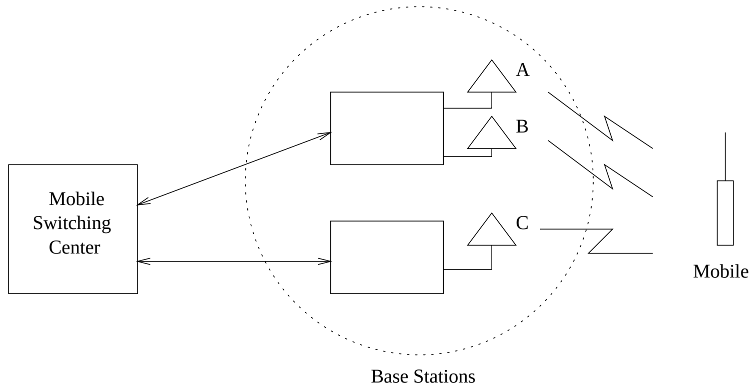

Figure 3: Channel Diversity in CDMA

ogy is widely considered to be the one with the best spectral efficiency and the best voice quality.

Most wireless systems support the notion of a hand-off, which means that the mobile can interact with a different antenna if it gets a better signal reception than one with which it is currently getting reception (each cell site may have many antennas). CDMA systems also support *soft* and *softer* hand-offs, in which a mobile can add multiple channels to a call in progress before dropping a channel due to poor reception. Channel diversity also improves sound quality in addition to making hand-offs proceed smoothly. Consider the case illustrated in Figure 3 where a mobile maintains channels through two antennas A and B sharing a physical connection to the mobile switching center, a scenario called softer hand-off. The mobile also participates in a soft hand-off by maintaining a channel through another antenna C with a different physical connection. Channel diversity complicates call processing, because state for multiple channels and the signal strength for each channel needs to be maintained for each call in progress. Recall also that each traffic channel requires a separate Walsh code, which are bounded shared resources. Moreover, actions taken in response to a signal strength change, also called a *trigger*, depend on current state, which is the result of the cumulative effect of previous triggers. For example, suppose antenna A in Figure 3 is the “primary” channel and antennas B and C are “secondary” channels. If signal reception for A drops to an exceedingly poor level, it causes a trigger. If C has currently the best signal, C becomes the new primary channel—an action called primary transfer. If the signal strength of A becomes high again later, channel A will be added to the call as a secondary channel. All these considerations makes testing through all possible hand-off scenarios a challenging task.

We performed testing on Lucent Technologies' CDMA call-processing system with two main goals. One, to ensure that as the signal strength of a mobile changes with respect to multiple antennas, it executes soft and softer hand-offs in a manner consistent with the system specification. Two, when multiple calls are alive, the Walsh-code allocation in the presence of hand-offs is done correctly, in the sense that no two channels are ever assigned conflicting (i.e., non-orthogonal) codes.

4. TESTING INFRASTRUCTURE

In order to achieve the above goals with VeriSoft, we need a model of the system's environment—the mobile phones and their activity—such that the behaviors of interest are exercised and checked. Moreover, we would like this environment model to generate the smallest state space possible that still contains these behaviors, so that verification time is not spent in exploring details that are not relevant to our goals. We first describe the overall testing infrastructure and then come back to the executable model we created for the environment.

We performed our testing in the context of a simulated wireless network environment, where the hardware for base stations, mobile switching centers and mobile phones is simulated to a level of detail that preserves all relevant message exchanges between these network elements. The call-processing software under test is run *unmodified from its product version* on the simulated platform[1]. In manual testing, a tester enters trigger commands in a shell that interprets and translates them to calls into the simulator. Actions generated by the system are reported to the user. For example, a tester may raise the signal power of another antenna, expecting to trigger the action of adding the antenna as a secondary channel to an ongoing call. The system informs the tester whether a hand-off took place, and optionally, which code was assigned to the new channel. Note that ignoring such a trigger is also a valid system response. If the tester finds the response acceptable, he/she repeats the process by trying another trigger based on the current state of the system.

VeriSoft automates test generation and execution in a seamless manner as described below. First, the simulation system is coupled with a test driver, which replaces the human in the description above. This test driver is a nondeterministic program that tries different triggers based on a set of nondeterministic choices. The test driver also maintains its view of the state of the system under test, and updates

[1]This infrastructure existed prior to this work.

```
Loop
  Pick a call: C
  Pick a (potential) channel for C: F
  Pick a new strength on F: S
  Send triggers for (C, F, S)
  Expect back response
     No response: ERROR
  Check for correctness of hand-off response
  Check for correctness of Walsh-code allocation
Endloop
```

Figure 4: Main Steps Executed by the Test Driver

this state based on the responses (or it reports an invalid response and terminates the execution of the current scenario). The pseudo-code in Figure 4 shows the main steps executed by the test driver.

Next, this test driver is run under VeriSoft control, which drives the test driver systematically and exhaustively through all paths in its state space. The primitive "Pick" in the above code is replaced by VeriSoft's `VS_toss(n)` call. Correctness checks are made using `VS_assert` calls.

The test driver just outlined is a model of the environment of the CDMA call-processing library. In order to limit the size of the state space being explored, a number of parameters are selected to have small but reasonable values. For example, only three discrete signal strengths are used—high, medium and low—since testing a more continuous range of signal strengths would increase the size of the state space but would have no bearing on the relevant behaviors of the system. In contrast, we do need to simulate a sufficiently large number of mobiles so that Walsh-code allocation is properly exercised. The test driver also needs enough flexibility to initiate and terminate calls dynamically. In summary, introducing nondeterminism where it is not relevant should be avoided whenever possible since it will increase the size of the state space unnecessarily.

The majority of work in creating this automated testing infrastructure is in encoding trigger-response behavior of the system under test in the test driver. Ideally, this behavior should be derived from requirements. In practice, we had to construct it by gathering information about the system from available documentation and by talking to local system architects, and in rare cases, by just trying it out on the real system "to see what happens". Fortunately, this is a one-time investment for several versions of the software, and for multiple hardware platforms, as long as the call-processing requirements are the same.

The test driver is implemented in about 1,500 lines of C code for the hand-off model, and another 2,000 lines of C++ code to verify the correctness of Walsh-code allocation. The latter program includes primitive classes from the actual product for performing basic operations on data structures encoding Walsh-code allocations, and is implemented as a separate Unix process for convenience (see below). We spent about three man-months of effort in creating this test driver

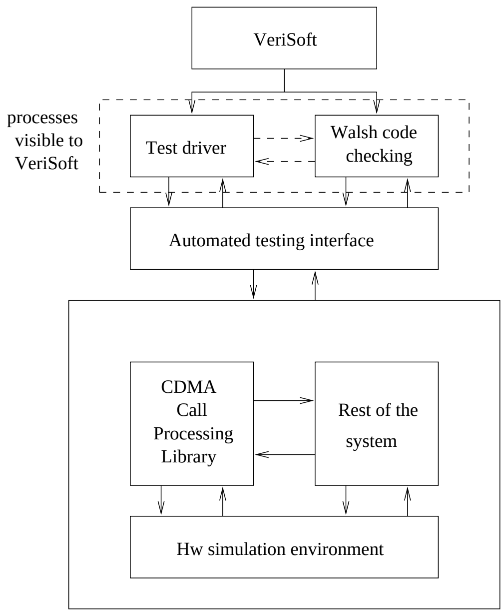

Figure 5: Architecture of Testing Infrastructure

and the underlying test-automation interface. A notable aspect of our approach is that it is possible to start testing with a partial test driver and refine it incrementally, rather than having to pay up-front the significant cost of creating a complete model of the system under test, as is required with traditional model checkers.

The overall architecture of the testing infrastructure is depicted in Figure 5. Only two processes are visible to VeriSoft: one nondeterministic C process implementing the pseudo-code described above and a second C++ process verifying the correctness of the current Walsh-code allocation. These two processes run on the same Unix machine and are synchronized using Unix semaphores which are also visible to VeriSoft. These two processes also interact with the CDMA product via a common testing interface, by sending and receiving strings of text, which correspond respectively to commands to and responses from the CDMA product. The latter is itself composed of the call-processing library and of several other components. All these components are implemented by many processes running on a simulation platform which is distributed across multiple machines connected in a network specially built for testing purposes.

In other words, the state space defined by this testing configuration is the "product" (i.e., the set of all possible concurrent executions) of the two processes visible to VeriSoft[2], which interact with each other and with the system under test whose processes themselves are hidden to VeriSoft. The latter simplification is needed to make an analysis of the CDMA product possible and tractable. As pointed out in Section 2, hiding possible sources of nondeterminisn to

[2]Thanks to the partial-order reduction algorithms used in VeriSoft, using two synchronized testing processes instead of one does not yield any increase in the number of paths in the state space.

VeriSoft has its price: race conditions due to invisible nondeterminism may be missed during state-space exploration, and reproducibility of scenarios leading to errors may also be compromised. Note that, during state-space exploration, VeriSoft records the type of visible operation performed during the execution of each transition in the state space; if, when re-executing a transition, VeriSoft observes that the type of that transition differ from the type recorded for that transition during a previous execution, it immediately reports an error signaling that the system being tested does not behave deterministically, which may be an indication that something is wrong in the system.

5. RESULTS

At the time of this writing, the testing infrastructure described in the previous section has been routinely used for over a year by the development organization in charge of Lucent's CDMA call-processing software. More than 1,500 runs of VeriSoft have been executed for systematically testing several software releases for several products versions (both 2G and 3G) for several target hardware platforms (three main types of base stations are commercialized by Lucent Technologies; the software for all three of them can be tested using the testing infrastructure described in this paper).

A vast majority of the VeriSoft runs were performed by members of the development team in charge of the CDMA call-processing library. Typically, when a new version of the software was available for testing, a member of the development team (often the third author of this paper) would run VeriSoft overnight in automatic state-space exploration mode, and then examine the next day erroneous scenarios using the interactive simulator and the several log files available to debug the entire product. Then, if the problem was not due to a software error in the testing infrastructure itself but looked like a real problem in the CDMA call-processing library (or in some other product component, or in the simulation environment), other developers with expertise in the parts of the software suspected to cause the problem would be contacted, usually via email or phone. In automatic state-space exploration mode, a single 12-hours (e.g., overnight) VeriSoft run typically generates, executes and evaluates millions of individuals tests grouped into thousands of call-processing scenarios (i.e, sequences of such tests).

It is worth noting that most VeriSoft runs so far were performed on software builds private to the development team in charge of the CDMA call-processing library, i.e., prior to the new software being submitted to be part of some official public build shared with other development and testing groups; this means that no defect-tracking tool was used to precisely record bugs found with an estimate of their severity (such tools are only used to track defects between official builds).

In the course of all these testing experiments, several previously unknown software bugs were found in the call-processing software, as well as in the simulation environment and the testing infrastructure. Problems found in the CDMA product were implementation errors, not errors in the high-level design of key algorithms themselves, which is not surprising since the core software being tested was already fairly mature at the beginning of this work. Due to space limitations and proprietary considerations, it is not possible to discuss here specific examples of bugs. In a nutshell, the most frequent types of errors found were improper initialization of variables or data structures, missing or broken case handlers in some states, and mismatched data formats across platforms. In some cases, the source of these errors could be traced to bug fixes applied inconsistently across configurations.

Although the above errors are standard types of coding errors, some of them were exposed only through very particular scenarios. Such errors would have been virtually impossible to detect using conventional testing techniques. The fact that VeriSoft detected errors that escaped traditional testing is not surprising, considering the following factors:

- Complex reactive systems such as CDMA call processing are notorious for exhibiting a very large number of different behaviors.
- Traditional testing is of limited help since test coverage is bound to be only a minute fraction of all possible behaviors of the system.
- Systematic state-space exploration can expose previously unknown bugs by exercising the system under test in enormously more possible ways.

Given commonly-used cost estimates for the difference between bugs found early versus late during the development process[3], the project reported in this paper has contributed significant savings to Lucent Technologies. VeriSoft also played a critical role in maintaining confidence in the correctness of the library as it evolved through several versions, both to add new features and to handle different kinds of cell sites.

Obviously, we cannot guarantee that the call-processing library is now entirely free of bugs. The goal of this work was to thoroughly test the library with respect to its handling of hand-offs and Walsh-code allocations. There could be other software defects, such as out-of-bounds references which did not have any effect in our tests, or bugs in other features of the library (e.g., the operator-administration interface) which were not exercised by VeriSoft at all. If desired, our infrastructure could be extended to test for these defects (additional test drivers for concurrently stimulating and checking other product interfaces can simply be composed in parallel to our current test driver under the control of VeriSoft).

[3] In the limit, debugging a single high-severity bug detected in the field can cost well over hundreds of thousands of dollars.

6. COSTS AND LIMITATIONS

Applying VeriSoft to systematically test the correctness of a concurrent reactive software application requires some effort. Here are the main costs and limitations associated with this approach to software model checking.

Test automation. Using VeriSoft for testing the correctness of a software product requires *test automation*, i.e., the ability to run and evaluate tests automatically. As testers know, developing a testing infrastructure that provides test automation can be in itself a significant effort. When test automation is already available, starting taking advantage of VeriSoft to significantly increase test coverage is usually easy since it may just involve modifying existing test scripts into nondeterministic ones and/or running multiple test scripts in parallel under the control of the VeriSoft scheduler.

Integration into testing environment. VeriSoft needs to be integrated into the execution environment of the system under test so it can control at run-time the execution of system processes. The primary task involved here is to declare which system calls of which processes are to be intercepted by VeriSoft and viewed as visible operations. Minimally, visible operations may simply include operations such as `VS_toss` and `VS_assert` for "black-box" testing of large applications. In the case of unit testing of applications containing only a handful of processes, system calls related to communication can also be declared as visible by mapping these to corresponding operations included in built-in VeriSoft libraries; for instance, sending a message (using whatever protocol is used by the application) can be mapped to a VeriSoft `send_to_queue` operation. Note that the actual system/protocol call is not replaced by `send_to_queue`, it is just annotated with the occurrence of that event. Mapping system calls related to communication to operations understood by VeriSoft can be tricky when complicated unusual communication objects are used. Instrumenting the execution itself can be done by overriding system calls at compile/link time, or via a binary-code or OS-kernel instrumentation, or the use of wrap-up functions intercepting events going in and out of the application being tested.

Test drivers. Like most model checkers, VeriSoft requires an executable representation of the environment of the system under test in order to drive its executions. Thanks to the `VS_toss` operation supported by VeriSoft, nondeterministic programs can be used as environment models (test drivers). Nondeterminism makes it possible to write very compact and elegant programs for generating large numbers of sequences of input events (test scenarios). Since the size of the state space depends on the amount of nondeterminism in the system, `VS_toss` should be used with care.

Specifying properties. Although VeriSoft can simply be used to detect standard errors such as segmentation faults, it is preferable to specify application-specific properties by means of assertions in test drivers in order to check the functional and behavioral correctness of the software application. Obviously, assertions previously inserted in the code itself by application developers can also be tested. Another possibility is to use tools (like Purify) that automatically insert assertions to check for standard programming errors such as memory leaks.

State explosion. The main practical limitation of VeriSoft, and of model checking in general, is the *state-explosion problem*: it is very easy for the user to define a state space that is too large to be explored exhaustively. State explosion can be controled by limiting the amount of nondeterminism visible to VeriSoft. Hiding nondeterminism due to concurrency inside the application being tested may result in errors being missed.

7. COMPARISON WITH RELATED WORK

Closely related work falls mostly into the following three broad categories.

7.1 Analysis-based Program Abstraction

As mentioned in the introduction, the other main approach to software model checking consists of (1) automatically extracting a model out of a software application by statically analyzing its code and abstracting away details, (2) applying traditional model-checking algorithms to analyze this abstract model, and then (3) mapping abstract counter-examples back to the code. The investigation of this approach can be traced back to early attempts to analyze concurrent programs written in concurrent programming languages such as Ada (e.g., [30, 26, 27, 9]). Other relevant work includes static analyses geared towards analyzing communication patterns in concurrent programs (e.g., [8, 11, 31]). Recently, several efforts have started aiming at providing model-checking tools based on source-code abstraction for mainstream popular programming languages such as C and Java. For instance, Bandera [10] can translate Java programs to the (finite-state) input languages of existing model checkers like SMV and SPIN, using user-guided abstraction, slicing and abstract interpretation techniques. SLAM [1] can translate sequential C programs to "boolean programs", which are essentially inter-procedural control-flow graphs extended with boolean variables, using an iterative automatic abstraction-refinement process based on the use of predicate abstraction and a specialized model-checking procedure. JavaPathFinder [32] can perform model checking of multi-threaded Java programs using a blend of static and dynamic program-analysis techniques. Although such tools have already been applied successfully to the analysis of several significant applications, their scope of applicability does not currently include applications as large and heterogeneous as the CDMA call-processing library.

7.2 Pattern-based Program Abstraction

Another related approach is Feaver [24], a tool and framework for translating annotated C programs into Promela, the input language of the model-checker SPIN. The abstraction performed by Feaver is specified by the user by defining pairs of C and Promela code patterns. When the C pattern is detected, the corresponding Promela pattern is generated as output in the abstract model. This simple approach to source-code abstraction is very general and flexible since translation rules can be provided for a multitude of program constructs and applications. On the negative side, specifying translation rules requires detailed knowledge of the source code and run-time architecture of the system being tested, as well as of Promela and model checking; moreover, no guarantees are provided about the soundness and completeness of the abstraction generated, which can be anything resulting from the translation rules specified by the user. Applying this approach to analyze the CDMA call-processing library software would have likely required several additional man-months of work to familiarize ourselves with the structure of the implementation code and the run-time architecture of the system and then write translation rules, in addition to the work needed to develop an executable representation of the system's environment and correctness properties; this option was never considered viable in our context since we did not have access to the developers whose expertise would have been needed to assist us in such a project.

7.3 Specification-based Testing

The closest alternative to the type of software model checking used in this work is perhaps specification-based testing frameworks for reactive programs (e.g., [33, 13, 29, 4, 25]). Given a specification of the input/output behavior of the system being tested represented by a finite-state machine (or a product of finite-state machines [16]) expressed in some modeling language, these techniques and tools can automatically generate a set of test sequences that cover the specification according to various coverage criteria. In contrast, VeriSoft generates test scenarios *dynamically* at run-time: state-space exploration is performed while the system is executing, and the outcome of previous test sequences (i.e., paths in the state space) typically influences the generation of following test sequences. Moreover, using VeriSoft does not require a specification of the input/output behavior of the system under test written in some specific FSM modeling language; instead, the environment of an open system can be represented by one or several processes executing arbitrary code, and the joint behavior of all these processes is then checked for "global" properties when exploring the resulting state space, in the style of what is usually done with model checking.

8. CONCLUSIONS

In our experience from this and several other projects with VeriSoft, whenever proper resources were allocated to a product-analysis project, we have *always* been able to find previously unknown bugs and provide valuable feedback to developers and testers. By attacking the testing problem through another angle, software model checking complements traditional testing, and can significantly contribute to increasing the confidence that a software product is ready to ship. We conclude with the following general observations on using software model checking in an industrial setting:

- Model checking is a simple testing/verification strategy. Used naively, the chances of getting interesting feed-back from using it might be small. Used properly, it can be extremely effective in *increasing test coverage*, *quickly detecting hard-to-find bugs*, and *reducing development intervals and costs*.
- Writing challenging tests and properties (i.e., those that might reveal important bugs) while limiting state-space explosion requires training, experience and some knowledge of how model checking works, as well as ingenuity and tenacity.
- In the end, whether or not to use software model checking is a question of economics: *how much do bugs cost*? For application domains where the cost of just a handful of high-severity bugs is high, software model checking is a valuable complement to code inspections and scenario-by-scenario requirement-driven testing. Applying software model checking using tools such as VeriSoft is not only technically feasible, but also economically sensible, since the cost required to apply model checking is likely to be significantly offset by the high cost (and embarrassment) of severe software failures in the field.

ACKNOWLEDGMENTS

We wish to thank the members of Lucent Technologies' Wireless Network Group who made this work possible by their help and support, with special thanks to Dave Ahnen, John Bamberger, Sharon Chen, Steve Meier, Antonio Ransom, and Steve Welsh.

VeriSoft can be downloaded from the web-site
`http://www.bell-labs.com/projects/verisoft`.

9. REFERENCES

[1] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, Paris, July 2001.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[3] 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.

[4] 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.

[5] 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.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[7] 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.

[8] C. Colby. Analyzing the communication topology of concurrent programs. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 202–213, New York, NY, USA, June 1995. ACM Press.

[9] 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.

[10] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[11] R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 214–225, New York, NY, USA, June 1995. ACM Press.

[12] 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.

[13] L. 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.

[14] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier/MIT Press, Amsterdam/Cambridge, 1990.

[15] J. 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.

[16] J.-C. Fernandez, C. Jard, T. Jeron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *Proc. 8th Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, August 1996. Springer-Verlag.

[17] 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.

[18] P. 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.

[19] 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.

[20] 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)*, pages 124–133, Clearwater Beach, March 1998.

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

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

[23] 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.

[24] G. J. Holzmann and M. H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of the 21st International Conference on Software Engineering*, pages 597–607, 1999.

[25] L. Jagadeesan, A. Porter, C. Puchol, J. 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.

[26] 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, Oct. 1991.

[27] 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.

[28] K. L. McMillan. *Symbolic Model Checking*. Kluwer

Academic Publishers, 1993.

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

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

[31] A. Venet. Abstract interpretation of the π-calculus. In M. Dam, editor, *Analysis and Verification of Multiple-Agent Languages (Proceedings of the Fifth LOMAPS Workshop)*, volume 1192 of *Lecture Notes in Computer Science*, pages 51–75. Springer-Verlag, 1997.

[32] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000 (15th International Conference on Automated Software Engineering)*, Grenoble, September 2000.

[33] M. Yannakakis and D. Lee. Testing Finite-State Machines. In *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing*, pages 476–485, 1991.