Experience Applying the SPIN Model Checker to an Industrial Telecommunications System

Barry Long, Juergen Dingel and T.C. Nicholas Graham
School of Computing
Queen's University
Kingston, Ontario, Canada, K7L 3N6
{dingel,graham}@cs.queensu.ca

ABSTRACT

Model checking has for years been advertised as a way of ensuring the correctness of complex software systems. However, there exist surprisingly few critical studies of the application of model checking to industrial-scale software systems by people other than the model checker's own authors. In this paper we report our experience in applying the Spin model checker to the validation of the failover protocols of a commercial telecommunications system. While we conclude that model checking is not yet ready for such applications, we find that current research in the model checking community is working to address the difficulties we encountered.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model Checking, Formal Methods; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical Verification, Specification Techniques

General Terms

Verification

Keywords

Experience report, model checking, formal methods

1. INTRODUCTION

Ensuring the correctness of telecommunication applications can be challenging, due to their complexity and inherent distribution and concurrency. In addition to traditional techniques such as testing, *model checking* has been viewed as a promising technique for validating the correctness of complex telecommunications software. The model checking approach involves creation of a mathematical model of the application, specifying desirable properties of the system using a temporal logic, and running a *model checker* to automatically verify that the properties hold of the model.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '08 Leipzig Germany
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Model checking promises several advantages over other approaches. Unlike testing, it is *comprehensive*, covering all possible execution sequences. Unlike traditional verification, it is *automatic*, as the model checker requires no human input. These features are particularly attractive for telecommunications applications in which problems of concurrency and distribution make traditional testing challenging.

Over the last decades, model checking has made impressive strides: it is now part of many undergraduate curricula around the world; it is explained in many text books written by experts in the field; it is implemented by many publically available tools, some of which are even open source; it is employed not only in many research groups, but also in many industrial settings, typically with the help of model checking experts; and it is the topic of hundreds of research papers. Despite these signs of increasing maturity, we note that:

- To the best of our knowledge, only a handful of studies have been published providing third-party evaluation of model checking on industrial scale systems. (Most evaluation of model checkers has been performed by the model checker's authors, and is usually restricted to toy applications.)

- Although it is difficult to assess the importance of model checking for industrial software development in general, it seems fair to say that large-scale industrial use of model checking is still elusive. The reason appears to be that many techniques work under restrictive assumptions which are invalidated by modern, industrial software.

This paper intends to improve this situation. First, we address the gap in the literature by providing a third-party evaluation of the application of a popular model checker to the analysis of the failover properties of a commercial web conferencing system. Second, we list the main problems that we encountered during our evaluation and give concrete suggestions to the model checking research community. More precisely, this paper reports on our work applying the SPIN model checker [13] to the analysis of the failover algorithms of WebArrow [11], a commercial web conferencing system. Like all telecommunications systems, WebArrow has high availability requirements, and therefore requires fast and reliable automatic repair when software, hardware and network failures occur. WebArrow's failover is implemented via complex distributed algorithms, and therefore is a good candidate for analysis via model checking.

The key results of this study were:

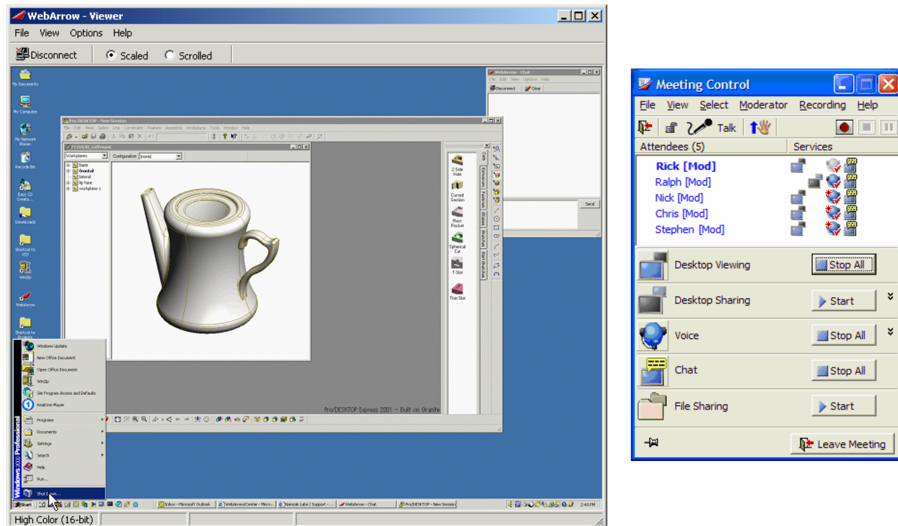

Figure 1: The WebArrow electronic conferencing system

- We found there to be a significant gap between the promises of model checking and the reality. Rather than enjoying fully automated checking of properties versus declarative system models, we battled problems of tractability in the checker. Overcoming these problems required iterative removal of detail from the system model and detailed knowledge of the algorithms used by the model checker. Ultimately, we were not sure that the version of the model that was checked corresponded closely enough to the real system to allow us to be confident in its correctness.

- The time to create and check the models was incompatible with realities of time-to-market demands.

- The level of documentation produced by developers, while sufficient for implementation, was not sufficient for creation of formal models. This greatly added to the effort of creating the models.

- In the end, only a handful of non-serious errors in the failover algorithms were found by the model checking technique.

The scarcity of independent evaluations in the software engineering research community has been lamented before [22]. Moreover, the need for more careful empirical evaluation of research results in formal analysis has also been identified recently [6]. Apart from sharing our experiences, a goal of this paper is to contribute to this trend by highlighting the importance of evaluating tools on industrial code.

The paper is organized as follows. We first introduce the WebArrow system and explain its failover protocols. We then detail our method for applying model checking to the analysis of failover in WebArrow. Finally, we present our results, and discuss how model checking might better support the validation of telecommunications systems.

2. THE PROBLEM

WebArrow [11] is a modern web-based telecommunications tool, allowing remote participants to meet using voice over IP, real-time collaborative document editing and textual chat. WebArrow operates as a standalone system, and has also been integrated into conference calling systems based on standard telephony. The WebArrow service is currently deployed across North America, Japan and India.

Customers treat WebArrow like a telephone, and demand similarly high availability of service. Therefore, a critical part of WebArrow's infrastructure is its support for detection and recovery from failure in computer hardware, software and network infrastructure.

Figure 1 shows an example of a WebArrow meeting from the point of view of one of its participants. In the main window, the participant can view and interact with a CAD application running on one of the other participant's computers. In a meeting control panel, the user sees an overview of the activities of other participants in the meeting, such as who is currently talking.

WebArrow meetings are scheduled via a web-based interface. Participants receive an invitation by email, and can join meetings by clicking on a link in that email without the need to install any software in advance.

2.1 WebArrow Architecture

WebArrow is deployed as a distributed system (figure 2). The system consists of two major components, with considerable redundancy. A *Meeting Management Server* (MMS) is responsible for scheduling meetings and arbitrating user entry to meetings. A *Conference Server* is used to host running meetings.

When participants receive a meeting invitation, they click on the included meeting URL to join the meeting. The URL is processed by the Meeting Management Server, itself running as an application under Apache. The Meeting Management Server selects a Conference Server to host the meeting, and directs the participant's client to connect to that server.

MMS servers are responsible for authentication of participants, downloading the WebArrow client to the participant's computer (via an ActiveX control), maintaining a list of available conference servers, allocating meetings to conference servers, and relocating meetings following failure of

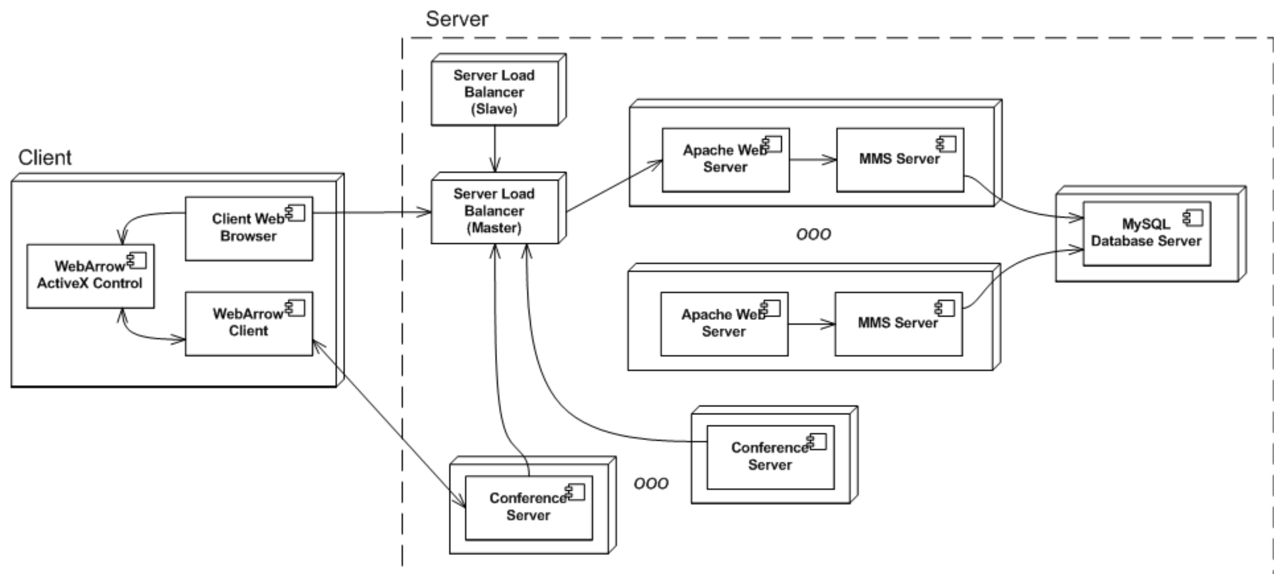

Figure 2: WebArrow deployment architecture

conference servers.

Conference servers host meetings. Screen, voice and chat data are sent from participants' clients to the conference server, where the data is mixed and sent to the participants. Conference servers periodically send load information to the MMS servers to support balanced allocation of meetings to conference servers.

WebArrow is implemented using a collection of languages, including C, C++, Perl, Java and JavaScript, running under Windows clients with Windows and Linux servers.

2.2 Failover in WebArrow

The architecture shown in figure 2 is prone to several kinds of failure. Any of the following components of the architecture may fail:

- An MMS server
- A conference server (possibly hosting a set of meetings)
- A client (or the client's network connection)
- A server in the database cluster
- The network connection between the conference server and MMS servers.

WebArrow combines a number of facilities for detecting and recovering from failures. Some of this technology is standard, available from third parties. The web-based MMS servers are organized in a standard sever farm, using a third-party load balancer to detect and decommission failed servers. A third-party MySQL database cluster is inherently fault tolerant.

Providing failover in clients and conference servers, however, requires complex, custom algorithms. Due to the distributed nature of WebArrow, each node has only partial information about the state of the system, increasing the difficulty of failure diagnosis and recovery. Each node in the system (client, MMS server, conference server) requires its own failover mechanisms, introducing the possibility of unintended interactions between them.

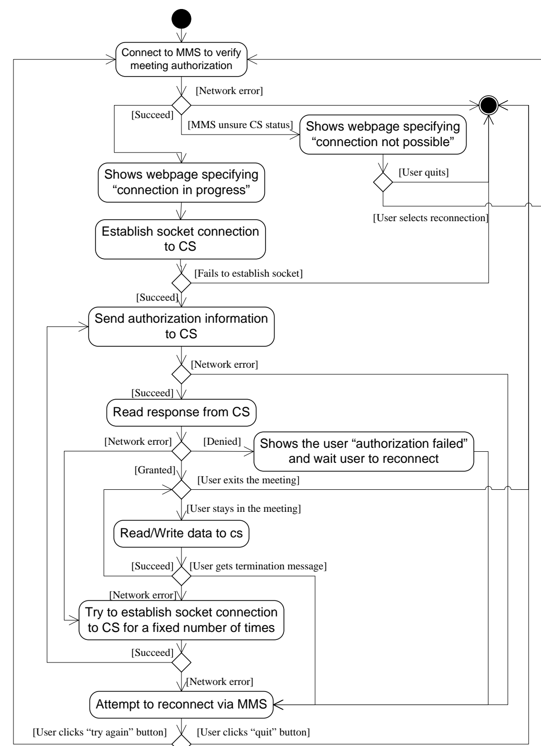

Figure 3: Activities of WebArrow client

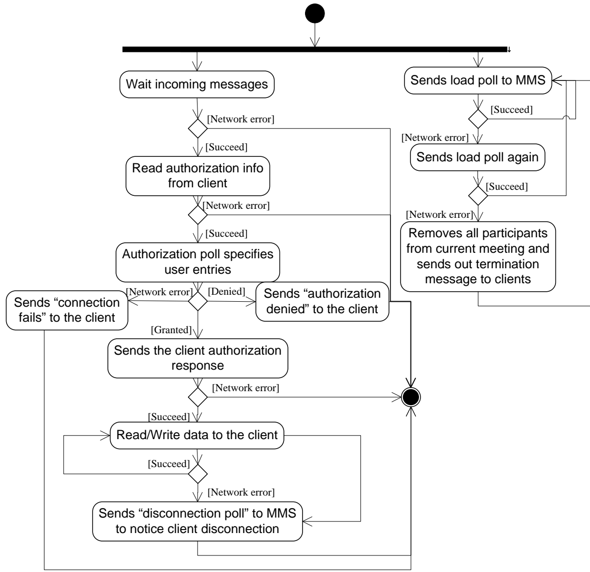

Figure 4: Activities of WebArrow conference server

For example, if a client loses its connection to the conference server, the problem may be that the conference server has failed, or that there is a network outage between the client and the conference server. Either kind of failure may be short-lived or persistent. Figure 3 shows the operation of the client, including how it handles failure. The client connects to a MMS server, validates itself, and is directed to a conference server. Once connected, the client enters a main loop of reading/writing data from/to the conference server. If the connection is lost, the client attempts to reestablish it. If reconnection is unsuccessful, the client falls back to requesting information from the MMS server, in hope that a new conference server has been assigned to this meeting. In this scenario, the client has no way of knowing whether the network or the conference server is at fault, and therefore must decide how to attempt to restore correct operation in the absence of complete information.

Figure 4 shows the operation of the conference server. Conference servers await connections from clients, validate those connections with the MMS server, and enter a main loop of reading/writing data from/to their clients. A single conference server may host multiple meetings and several hundred clients. Meanwhile, the conference server sends load information to the MMS server. These load polls help the MMS server in load-balancing a set of conference servers, and also act as "alive" messages. If two load polls in a row are missed, the MMS server assumes that the conference server has failed, and re-allocates its meetings to other conference servers. If a conference server is running but fails to send two load polls in a row (due to network error), it knows that the MMS server will consider it to have failed. To maintain consistency over the system, the conference server must then act as if it has indeed failed, and direct its clients to reconnect to the MMS server.

It took approximately 12 staff-months at Namzak Labs to design and implement the conference server's failover mechanisms. Despite the fact that the failover code had been deployed in production and was apparently working, its authors were not completely convinced of its correctness, largely because it is difficult to use traditional testing to fully exercise all possible paths through a distributed system of this complexity. We therefore deemed it an interesting experiment to use formal modeling and model-checking to verify the correctness of WebArrow's failover mechanisms.

The use of formal methods could not be considered a great success. More time was required to carry out the formal analysis than had been invested in developing the failover mechanisms themselves, and no serious errors were found. Such a negative result would be a success if we could have strongly concluded that WebArrow's algorithms were correct. However, to make make model checking tractable, our models were so simplified that we remain unsure whether serious errors remain in WebArrow's failover algorithms.

As we will see in section 4, we are able to draw several conclusions about the root causes of our difficulties, which we hope will provide useful direction to those carrying out research in model checking techniques. Happily, many of our conclusions validate the current directions in model checking research.

We now present the method used in formally analyzing WebArrow's failover mechanisms, following which we present our results and analysis.

3. METHOD

In this section, we summarize our experimental methodology, sketching the sequence of steps followed during the experiment. We outline some of the properties we checked during our analysis. Finally, we describe the experimental platform.

3.1 Experimental Methodology

We used the Spin model checker for our work, because it is known as a mature, well-documented tool and as one of the most heavily optimized model checkers available. The Promela language seemed to be a good fit with our intent to model WebArrow mainly at the level of inter-process messages. Finally, more recent and "modern" approaches supporting, e.g., model extraction or model checking of C or Java code did not appear to be as well documented [14], or required the entire system to be implemented in a single language or run on a single node.

We broke our modeling and analysis effort into the following eight steps:

1. Summarize failover protocol with UML activity diagrams: With the help of technical documents and interviews with the developers of WebArrow, the failover protocols were summarized using UML activity diagrams. Drafts of the diagrams were refined based on discussions with developers until the developers assured us of their correctness and adequateness. The activity diagrams for WebArrow clients and conference servers are shown in figures 3 and 4, respectively.

2. Build Promela model: We built a Promela model from the activity diagrams. The behaviour of the model was checked using Spin's random and interactive simulation capabilities. Whenever the simulation revealed a modeling error, the Promela model was corrected. Whenever the simulation revealed a misunderstanding of the failover protocol, we returned to Step 1 and modified the activity diagrams. It is also possible for

the simulation step to reveal a defect in the original system; however, this did not happen in our experiments.

3. Check for deadlock: The Promela model was checked for deadlock using Spin. If a deadlock was found, the trace was inspected to locate the defect in the Promela code. Then, a change to fix the defect was made either to the activity diagram or the Promela model.

4. Express properties in LTL: Desirable properties of the failover protocol were collected and expressed in Linear Temporal Logic (LTL). Quite often, the introduction of auxiliary variables was necessary to make certain aspects of the execution observable.

5. Verify LTL properties: We used the Spin model checker to determine whether the Promela model satisfied the LTL properties.

6. Optimize and simplify the model: Often, the verification of our most detailed model could not be completed due to state space explosion. To address this problem, we applied optimizations to the Promela code (e.g., restricting variable types and using auxiliary variables) and to simplify the model in various ways (e.g., by restricting number of clients, or the types of failures). We thus ended up with a sequence of four models in which the first model was the most simplified and the last model was the most detailed. The simplifications were chosen to ensure that an analysis of the simplified model was sound with respect to property violations: if a simplified model violates a property, then the unsimplified model does, too. Note, however, that we did not prove this relationship formally. Verification of a given LTL property would start with the most simplified model; if an exhaustive analysis of that model was possible, we attempted to analyse the next model in the sequence.

7. Debug model using counter-examples: If Spin found a property to be violated, we inspected the counter example. If the property was expected to hold, or the counter example revealed an additional, unexpected violation, we had to correct the Promela code, forcing us to return to Step 2. On occasion, the counter examples turned out to be "spurious" due to, for instance, the incorrect update of auxiliary variables or "unexpected" features of Promela such as the fact that no fairness constraint is imposed on the selection of guards in do-loops.

8. Record results: The analysis result was recorded.

3.2 Properties

In cooperation with WebArrow's developers, we first collected system requirements informally. We then attempted to formalize as many of these requirements as possible in Linear Temporal Logic (LTL). This formalization was substantially complicated by the fact that components in a distributed system typically only have partial knowledge of the global state, and that the transmission of a message may fail or be delayed arbitrarily. Consequently, in the absence of a global clock, even formally specifying what it means for the global state to be consistent is difficult. Moreover, the reestablishment of a consistent global state may be delayed arbitrarily and possibly indefinitely. As mentioned before, the formalization often required the introduction of auxiliary variables, for example, to indicate that a specific line of code is about to be executed.

Our formalization resulted in 12 formal properties, loosely grouped into three categories: (1) properties formulated from the perspective of a client, (2) properties formulated from the perspective of the MMS, and (3) properties formulated from a global perspective. Sample properties for each category follow:

1. Sample properties from perspective of client:
   - *"Once a client receives an authorization, it always eventually enters meeting"* (Property 2)
   - *"Every client is always connected to at most one conference server"* (Property 10)

2. Sample property from perspective of MMS:
   - *"Every client is in at most one meeting at a time"* (Property 4)

3. Sample property from global perspective:
   - *"The system never deadlocks"* (Property 0)
   - *"When a client leaves a meeting, the MMS will eventually update its records accordingly"* (Property 5)
   - *"If a client has left a meeting, the MMS server will eventually believe that the client is not in the meeting"* (Property 6)

Not all properties are applicable to all models; for instance, Property 10 is vacuous in models with fewer than two conference servers. Moreover, not all properties were expected to be always satisfied; that is, for some properties the analysis was expected to produce specific counter examples. In these cases, the purpose of the analysis was to ensure that the system exhibited exactly the expected counter examples and not any others. Consider, for instance, Property 2. In models allowing client failure, this property will be violated in executions during which a client fails after it has received an authorization, but before it enters a meeting.

3.3 Model Simplifications and Promela Optimizations

The judicious use of optimizations is often critical to the successful use of model checking. A balance between detail and tractability must be found, typically through a lengthy, tedious, and unscientific process of trial-and-error. And despite all efforts, significant resources may still be required. Our experiment supports this observation: we tried many, many approaches to abstracting our model and optimizing our analyses. Despite this, our most comprehensive model still is relatively small and substantially simplified, and its analysis required over 2.5GB of memory and about 25 hours to run.

3.4 Model simplifications

To establish a base line, we started by considering a system without failure. This led to models that we named M1, M2, and M3. Besides the MMS, M1 had one client and one

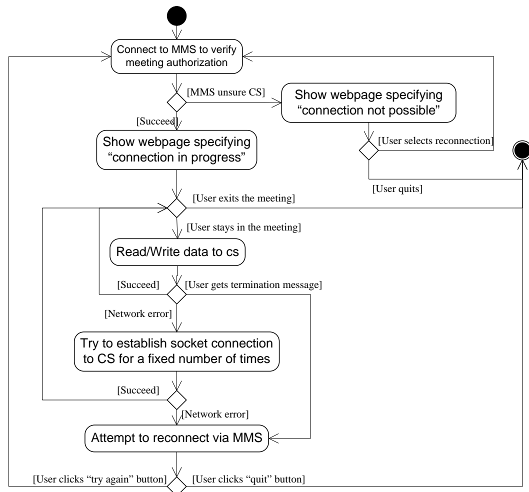

Figure 5: Model M4 of the WebArrow client

conference server. M2 had 1 client and 2 conference servers and M3 had 2 clients and 2 conference servers. Our initial attempts to include failure into model M3 quickly led to intractable execution. After many attempts to balance detail and tractability, we arrived at model M4. The conference server part of M4 is shown in figure 5. As can be seen, the steps of establishing sockets (Step 4 in figure 3) and authorization (Steps 5 to 7 in figure 3) not only are kept failure-free, but have been removed entirely. After discussions with WebArrow developers, we decided that these steps are less relevant to the failover protocol than those steps which are critical to interactions between nodes and removed them to reduce the state space. Additionally, the corresponding steps in the conference server, such as Steps 2 to 6 in Figure 4, were also removed. Model M4 comprised approximately 700 lines of Promela code (including comments).

3.5 Spin optimizations

Apart from the built-in optimizations offered by Spin (e.g., collapse compression, POR, minimization, bitstate hashing), we had to apply the following "custom" optimizations:

- Implementation of channels: Our initial decision to implement communication in WebArrow through separate processes which encapsulate the implementation of failure, quickly led to intractable models. Instead, standard Promela channels of capacity 1 were used and failure was indicated through a distinguished value.

- Manual statement merging: Whenever the intermediate states passed through the execution of a sequence of statements in a process was unobservable by its environment, it is safe to execute that sequence in one atomic step. This applies, for instance, to updates to process-local variables (an optimization which Spin already performs automatically), but in some situations also to global variables.

- Minimize variable types: To allow for a maximally compact state representation, it was often necessary to minimize the types of variables (e.g., use "bool" instead of "byte").

3.6 Experimental Platform

The verification was performed using version 4.2.6 of the Spin model checker. We used a Sun Fire V65X Linux machine with four 3.06GHz Xeon CPUs and 5GB memory, and a Windows XP Pentium 4 machine with a 3.2 GHz CPU and 2.5GB memory. The small machine was used to execute preliminary tests while full verifications were attempted on the large machine.

4. RESULTS AND ANALYSIS

The analysis only produced three previously unknown issues. All of these give rise to temporarily degraded usability rather than malfunction, and we thus consider them "non-serious". In two cases, the root cause was an "oscillating" network connection, that is, a network connection that rapidly and repeatedly changes between being up and down. In the other case, a race condition led to temporary inconsistency.

4.1 Oscillating network connections

Our Spin analysis found a counter example to Property 5. In this counter example, there is an oscillating network connection between a client and the conference server which is hosting a meeting. First, the client loses the connection to the conference server. After trying to connect to the conference server for a fixed number of times, the client assumes that the conference server has failed. The client then connects to the MMS server to ask for connection information. However, since the connection between the conference server and the MMS server works well, the MMS server sends a message to the client and confirms that the meeting is still running on that conference server. Meanwhile, the connection between the client and the conference server recovers and the client successfully connects to the conference server. However, as the connection oscillates, the sequence above repeats. This counter example is depicted in figure 6. The counter example found for Property 6 is similar. Here, the network connection between the conference server and the MMS oscillates.

We find that in case of these kinds of oscillating network connections, WebArrow's behavior is correct, but may lead to poor usability. While these oscillating connections are certainly possible, they are unlikely in practise, particularly because the conference and MMS servers are typically colocated.

4.2 Temporary inconsistency

A third scenario our analysis discovered results in a state inconsistency between the client and the MMS. First, a client leaves a meeting and sends a quit message to the conference server. However, the conference server's load poll does not arrive at the MMS. Consequently, the MMS server still believes the client is in the meeting. If the client decides to join the meeting again, it will receive an error indicating that it is in the meeting now. However, this state inconsistency is temporary, because as soon as the poll arrives from the conference server, the MMS becomes up to date. Moreover, the conditions under which this problem arises are rare, and since the failover protocols resolve the problem in time, it is also not considered serious.

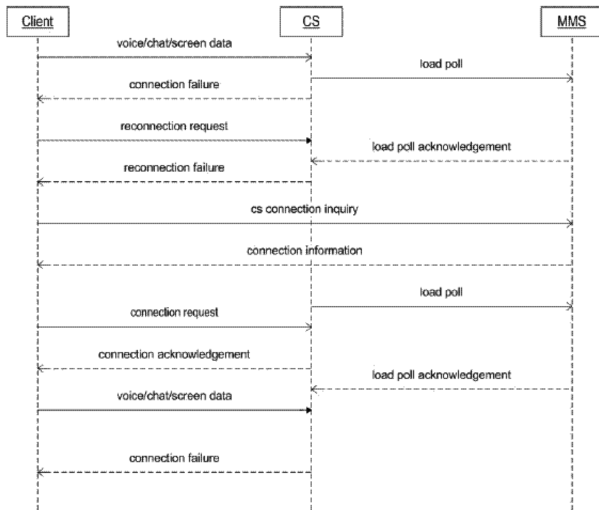

Figure 6: Sequence diagram of the counter example for Property 5 found by Spin

4.3 What the analysis told us about the use of model checking

Our use of Spin to model and check WebArrow's failover protocols reveals several issues in the state of the art in model checking.

4.3.1 *Big mismatch between development and analysis models*

While the WebArrow developers did use some models during development, the level of detail required for our Spin analysis vastly exceeded that of the available models. This was particularly true when we attempted to determine system properties (as described in section 3.2): WebArrow's developers clearly understood what it meant for the system to behave correctly, but had never attempted to encode this behaviour in terms of declarative properties.

4.3.2 *Big gap between promise of model checking and reality*

Model checking is sometimes described as a "push-button" technology that in cases where an exhaustive analysis is possible, provides "complete certainty". Others have already observed that this is hardly an adequate description of model checking [9]. Our experience supports that observation.

- "push-button": The successful use of model checking on medium- to large-sized systems requires substantial expertise about the system to be analyzed and about the model checker. In our case, many face-to-face interactions were necessary to "extract" the relevant expertise from the WebArrow developers. Moreover, knowledge about, e.g., the algorithms and optimizations underlying Spin was necessary to be able to use Spin properly and to be able to develop the necessary intuition about which aspects of a model cause state explosion and to find ways to avoid them without simplifying the model unnecessarily.

- "certainty": As described in section 3, we were forced to significantly simplify the model. As a result, the exact relationship between the model and the real code became increasingly unclear. Arguments for the soundness of a certain simplification often were very informal. In the end, we felt we did not know exactly anymore what an analysis result meant for the original WebArrow system and that we had thus lost one of the central characteristics of a sound, formal analysis [6]. For instance, we cannot rule out that a positive analysis result (i.e., one that is exhaustive, but does not find any property violations), was a "false positive" (i.e., the original system does violate the property, while the analyzed model does not). Examples of some of our simplifications that may lead to these false positives include: restricting the number of clients, limiting the channel capacity to one, and removing certain kinds of failure in model M4. Moreover, false negatives can always be introduced through bugs in the model (e.g., the incorrect update of an auxiliary variable). While the potential for false negatives can be reduced by thoroughly analyzing the counter examples with the help of the developers, they also cannot be completely ruled out.

4.3.3 *Resources exceed tolerance of average industrial software developers*

The work was carried out by an MSc student under close supervision of two faculty members, one of whom has devoted his career to formal methods. Overall, the time to learn Promela and Spin, to extract the knowledge necessary knowledge from WebArrow developers, and to create, simplify and analyze the models exceeded one person-year. Since according to the WebArrow developers, the realization of WebArrow's failover mechanisms only required about one person-year, attempting to incorporate our analyses into the development cycle would have resulted in an unacceptable delay.

The most time-intensive activities in our experiment included the construction of the models due to the many face-to-face meetings with the developers that proved necessary, and the trial and error process necessary to find a detailed, yet tractable model. Moreover, the verification runs were very resource-intensive. For instance, the analyses for model M4 required about 2.7GB RAM and approximately 25 hours.

Shorter activities that nonetheless contributed substantially to the overall time required include: setting up the analysis of certain properties by manually adding necessary auxiliary variables to the model and removing unnecessary ones; the use of user-guided simulation, because input of user choices in XSpin must be performed manually which is tedious and error-prone; and the analysis of counter examples, because the generated counter-examples were often extremely long.

4.4 Lessons learned for model checking community

The lessons identified in the previous section present a sobering view of the practicality of using model checking for telecommunications applications such as WebArrow, that is, applications involving distribution, multiple implementation languages, and complex algorithmic interaction between nodes. It is important to note that this experiment was in a sense modest: we restricted our analysis to WebArrow's failover algorithms only, a small part of its complete

functionality.

We are able, however, to draw useful directions from this experiment that we hope will be helpful to researchers in the model checking community. Several of our conclusions in fact validate current research directions in the field.

4.4.1 Work on independent evaluations critical

One of the main things that we learned during this work is that, to the best of our knowledge, not very many studies have been published in which model checking was used for the analysis of industrial-strength software systems and in which none of the authors of the study was intimately familiar with the model checker used (e.g., because they have contributed to its development). There are some notable exceptions [21, 2, 16, 15], which we will discuss in section 5. In other words, there is severe a lack of independent evaluation of the use of model checking on industrial software. While this lack appears to be part of a more general phenomenon affecting much of software engineering research [22], it needs attention nonetheless, because independent evaluations can contribute substantially to the successful transfer of research into practice, by helping to identify new research issues worthy of attention. For instance, we find that more research needs to be done to facilitate and streamline the use of model checking and to allow non-experts to use it effectively and within reasonable resource limits. Moreover, as discussed in more detail below, model checking currently does not seem equipped to handle the heterogeneity of modern software applications.

4.4.2 Work on tools critical

Efforts should be made to increase usability of checking tools. For instance, more of the complexity of model checking should be hidden behind an interface. Spin already offers many useful features (e.g., both GUI-operation and command-line operation are possible allowing for, e.g., the graphical display of counter examples and batch job processing), but more could be done.

For instance, verification tool builders might learn from the current trend to "adaptive" and "autonomic" computing in DBMSs, and devise ways to automatically determine optimal choices for model checker settings.

Furthermore, it might help to allow a model checker to be used in different modes according to the expertise of its user (e.g., a "novice mode" with simplified interaction using default or heuristic values, a "superuser mode" in which the user can see and customize every aspect of the behaviour of the tool, and an "intermediate mode" which strikes a balance).

Finally, better support for repeated, large-scale experimentation would be useful. For instance, the execution of many analyses in which only a few parameters change each time could be simplified through the use of "session files" which package all parameter and user settings. Also, providing the input required for user-guided simulation could be facilitated through, e.g., the graphical animation and selection of choices.

4.4.3 Reduce "semantic gap" between program and model

As mentioned in section 4.3, the gap between the original WebArrow system and the model M4 is large, to the point that we fear that errors in WebArrow's protocols may have been abstracted way. Our inability to achieve comprehensive analyses of our larger models made this necessary.

As a response, at least two research directions seem important: first, the capabilities for automatic model extraction, optimization, and traceability could be improved. Research work that is pursuing this direction include the Bandera tool [5] which aims at model checking Java code through an automatic translation to finite state machines and the FeaVer tool [14] which allows for an automatic translation of C code to Promela via a user-supplied translation table. FeaVer allows counter examples to be presented in terms of the original, untranslated model and thus supports traceability; Bandera currently does not. Both approaches also feature automatic optimization techniques. However, as we have seen during our work, there clearly is the need for more. It should be noted, though, that automatically extracted models will typically be larger than those created by hand, exacerbating issues of tractability.

Second, if more incomplete analyses are acceptable, more detailed models could be used. Recent work on software model checking [8, 20, 17] pursues this direction by attempting to interpret the code as the model. Here, the checker turns into a sophisticated testing tool able to show incorrectness by finding bugs, but often unable to show correctness since searches remain incomplete. Maximizing the coverage of incomplete searches and their ability to find bugs are important research goals. Recent work on, e.g., improving random testing by combining it with static analysis, symbolic execution, and constraint solving appears promising [10, 18].

4.4.4 Deal with heterogeneity of modern applications

WebArrow is typical of telecommunications applications in that it runs as a distributed system and is composed of components written in a variety of programming languages. While the research we described above is using automated model extraction to help close the "semantic gap" between code and model, none of the existing tools can deal with these realities of multi-platform, multi-node environments. Considerable research is required to address this problem.

4.4.5 Address mismatch between models used during development those required for model checking

As we discussed in section 3, the documentation and models of WebArrow available to us at the outset of the project were inadequate for building formal models in Spin. Developers document their systems to a level that allows implementation and testing. In our experience, this documentation did not have sufficient detail to allow a formal model to be constructed.

This was particularly the case when expressing properties of the system, where developers were not used to thinking of properties in terms of global invariants. If model checking is going to be practically integrated into the software life cycles of telecommunications applications, considerable thought is required as to what documentation should be created, at what point in the process, and by whom.

We conclude that our experiment has shown useful directions for further research in the model checking community. We are happy that our experience validates some ongoing research directions, particularly the automated extraction of models from code. Our experience highlights other areas that are receiving less attention, such as the ability to deal

with distribution (and its inherent partial knowledge) and systems built using multiple languages.

5. RELATED WORK

The research literature contains many papers documenting the successful use of model checking in general (including, e.g. [4, 12, 3]). However, it is considerably harder to find studies in which model checking was applied to industrial-strength software systems and where none of the authors of the study was a developer of the model checker being used. We have been able to find only four exceptions [21, 2, 16, 15]. In all these cases, quite detailed knowledge of the system was required to be able to extract sufficiently detailed, yet tractable models. All papers report that previously unknown bugs were found and are quite positive about the use of model checking as an analysis technique for industrial-strength software. Although the papers do not contain enough information about, e.g., the resources used and the background of the participants, it appears quite likely that the total effort required would have overwhelmed any average industrial software development project. In other words, while these papers do provide evidence for the usefulness of model checking for industrial software development, they do not, in our opinion, show that large-scale adoption of model checking in industry is currently feasible.

To see how far removed software model checking is from that goal, it is instructive to look at some of the more recent successful uses of model checking on industrial-size code such as the PathStar project [14], the DEOS project [19], the SLAM project [1], and the TCP/IP project [7]. For each project, we will attempt to convey the effort involved by briefly sketching how two of the most critical tasks in model checking were dealt with: model extraction and state space explosion. For details on the projects themselves, the reader is referred to the cited papers.

PathStar: In the PathStar project, Lucent's PathStar access server was analyzed with respect to feature requirements. The tool built for this purpose consisted of an editor for the graphical input of temporal properties, a user-supplied mapping table for the translation of C code to PROMELA code, a dedicated multi-server infrastructure for concurrent analysis, and a web-based interface for monitoring and controlling of the verification runs which were performed by Spin. "Proof approximation techniques" were used to generate verification runs of varying thoroughness. The table-driven translation was complemented by the manual replacement of expressions by, e.g., non-deterministic choice among a set of values guided by the "relevance" of the data. Counter examples were analyzed manually and the verification was monitored for anomalous behaviour to detect vacuous and thus buggy properties.

DEOS: The goal of the DEOS project was to analyze the DEOS real-time scheduling kernel developed by Honeywell in C++. Initially, the Spin model checker was used on a manually constructed Promela model, but later a source-level model checker for Java called Java Pathfinder [20] was built and applied to a manual reimplementation of DEOS in Java, because the manual C++-to-Promela translation was deemed "not practical". Abstractions of the real-time and object-oriented aspects of the system were developed and applied manually. Moreover, both manual and automatic environment generation processes using LTL properties were developed. Due to overapproximation, the actual verification process required the manual analysis of any counter examples and a corresponding manual refinement of the model when the counter example was found to be spurious.

SLAM: The goal of the Slam project was to analyze Windows driver code written in C with respect to a limited set of "interface usage rules". Slam managed to automate the abstract-check-refine loop also used in the DEOS and the PathStar projects completely. A model was extracted from the code using predicate abstraction. After a model checking run, counter examples were analyzed for feasibility. New predicates for the refinement of the model were then "discovered" from infeasible counter examples using a heuristic process. All these steps were carried out automatically. The Slam technology is now offered to Windows device driver developers as SDV (Static Driver Verifier).

TCP/IP: In the TCP/IP project, the Linux TCP/IP protocol implementation was analyzed. A new source-level model checker CMC was built alleviating the model extraction problem [17]. However, the standard attempt to separate the system from its environment by discovering the interface and "stubbing" out calls to the environment appropriately failed "after spending months". Instead, an abstraction of the entire Linux kernel was fed through the model checker. Among the search optimizations developed by the team to make this feasible were heuristics to steer the search towards more interesting behaviours. More precisely, transitions that led to states that were very different from the previous state were deemed more interesting and examined first.

All four projects are impressive in their goals and accomplishments. However, we must also conclude that all consumed large amounts of resources, required detailed domain knowledge of the system to be analyzed (often including access to the system's developers), and enough expertise in model checking and program analysis to be able to solve open research problems such as the development of new search heuristics and abstractions and understanding their sometimes extremely subtle impact on, e.g., performance and soundness. In light of this, we do not share the optimism expressed by members of the SLAM team at the beginning of 2004 [1]:

> "Beyond SLAM and SDV, we predict that in the next five years we will see partial specifications and associated checking tools widely used within the software industry."

Much more work is needed, e.g., to bring the ideas developed above into the mainstream and to encapsulate them in usable tools of sufficient generality, before analyses such as these can become routine industrial practise.

6. CONCLUSION

This paper has reported on our experience in using the Spin model checker to validate the correctness of the failover mechanisms in the WebArrow electronic conferencing system. Our conclusion is that model checking technology is not yet sufficiently mature for use in the development of telecommunications systems such as WebArrow. The central problems were the difficulty of creating models which could be validated with a tractable set of resources, the semantic gap between the models we were able to validate

and the code implementing those models, the difficulty of creating models of systems involving distribution and multiple implementation technologies, and the time and expertise required to perform model checking.

These issues have to some extent already been identified within the model checking community, and ongoing research is attempting to address them. Our experience therefore helps confirm the direction of current research, while emphasizing areas that are currently not receiving attention in the research community.

7. ACKNOWLEDGMENTS

We would like to thanks Chris Walmsley of Namzak Labs for his help in understanding WebArrow's failover protocols. The authors gratefully acknowledge the support of the Natural Science and Engineering Research Council (NSERC).

8. REFERENCES

[1] T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft Research, January 2004.

[2] B. Boigelot and P. Godefroid. Model checking in practise: An analysis of the access.bus protocol using spin. In *Formal Methods Europe (FME)*, LNCS 1051, pages 465–478. Springer Verlag, 1996.

[3] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: An industrial case study. In *International Conference on Software Engineering (ICSE 2002)*, pages 431–441, May 2002.

[4] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[5] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *23rd International Conference on Software Engineering (ICSE'01)*, May 2001.

[6] M. Dwyer, J. Hatcliff, Robby, C. Pasareanu, and W. Visser. Formal software analysis: Emerging trends in software model checking. In *28th International Conference on Software Engineering, Track on Future of Software Engineering (ICSE FoSE 2007)*, pages 120–136. ACM Press, 2007.

[7] D. Engler and M. Musuvathi. Model-checking large network protocol implementations. In *First Conference on Network System Design and Implementation (NSDI)*, 2004.

[8] P. Godefroid. Software model checking: The Verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[9] P. Godefroid. Software model checking via static and dynamic program analysis (invited tutorial). Summer School on MOdelling and VErifying parallel Processes (MOVEP'06), June 2006.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, Chicago, June 2005.

[11] T. Graham, R. Kazman, and C. Walmsley. Agility and experimentation: Practical techniques for resolving architectural tradeoffs. In *ICSE '07*, pages 519–528, 2007.

[12] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal analysis of the remote agent before and after flight. In *5th NASA Langley Formal Methods Workshop*, June 2000.

[13] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[14] G. Holzmann and M. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2002. Special Issue: Software.

[15] G. Hughesa, S. Rajana, and T. Sidlea. Error detection in concurrent Java programs. In *Workshop on Software Model Checking (SoftMC 2005)*, Electronic Notes in Theoretical Computer Science, pages 45–58, February 2006.

[16] J. Ivers. Lessons learned model checking an industrial communication library. Technical Report CMU/SEI-2005-TN-039, Carnegie Mellon University, 2005.

[17] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Symposium on Operating Systems Design and Implementation (OSDI*, 2002.

[18] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE)*, Minneapolis, May 2007.

[19] J. Penix, W. Visser, S. Park, C. Pasareanu, E. Engstrom, A. Larson, and N. Weininger. Verifying time partitioning in the deos scheduling kernel. *Formal Methods in System Design*, 26(2):103–135, 2005.

[20] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[21] J. Wing and M. Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, 28:273–299, 1997.

[22] C. Zannier, G. Melnik, and F. Maurer. On the success of empirical studies in the International Conference on Software Engineering. In *28th International Conference on Software Engineering (ICSE 2006)*, pages 341–350. ACM Press, May 2006.