Formal Verification of Concurrent Software:
Two Case Studies

[Extended Abstract] *

Hana Chockler Eitan Farchi
Ziv Glazberg Benny Godlin
Yarden Nir-Buchbinder
IBM Haifa Research Laboratories
Haifa University, Mount Carmel
Haifa 31905, Israel
hanac,farchi,glazberg,godlin,yarden@il.ibm.com

Ishai Rabinovitz †
Mellanox Technologies
Israel
ishair@gmail.com

ABSTRACT
Software model checking came to the focus of research recently, with sharp growth in the number of safety-critical applications and in the complexity of software. In model checking of software we meet some obstacles which do not exist in model checking of hardware: the state explosion problem is more acute, the model often consists of many processes that run concurrently, and there also can be a requirement for correct behavior in the presence of failures. Also, the programs are written in high-level programming languages, which causes two problems: the model-checker has to understand a programming language, and the state space of the program can be infinite (for example, the range of values for integer values is infinite in theory). In this paper, we present two case studies of real projects at IBM, which were formally modeled and verified using ExpliSAT model checker, as well a testing tool ConTest. The second case also involves modeling limited Byzantine (malicious) failures of processes. We discuss the special structure of the control flow graph of these programs and perform probabilistic analysis of the number of random executions needed in order to execute all control flow paths with high probability. We also compare the performance of ConTest and ExpliSAT on these case studies.

Categories and Subject Descriptors
D.2 [Software Engineering]: Software/Program Verifica-
 tion; F.3 [Logics and Meanings of Programs]: Specify-
ing and Verifying and Reasoning about Programs

General Terms
Verification

1. INTRODUCTION
Model checking is an automatic method of checking the correctness of computerized systems [3, 14, 9]. It checks whether a formally defined model of the verified system satisfies its specification, which is also described using some formal language, for example, temporal logic [13]. If model checking succeeds, the system is shown to be correct with respect to the specification. If a model checker discovers an error, it outputs a trace of an erroneous execution. Model checking tools are steadily gaining popularity in hardware verification, where bugs are costly to repair and systems are usually small.

In the last years, there has been growing awareness of the need for formal verification of software systems (see [7] for a survey). Since software is often deployed in safety critical applications, its correctness and reliability have become issues of utmost importance. Recently, Tony Hoare put the challenge of creating a verifying compiler among the greatest challenges of computing research [8]. His definition of a verifying compiler is very close to the definition of a model checker: it uses formally defined properties in order to mathematically and logically reason about the correctness of a given computer program in some high-level programming language.

One of the most challenging areas in software verification is verification of concurrent systems. The large number of possible interleavings of concurrently executing processes makes the so-called “state-explosion problem” especially acute. Another problem of concurrent systems is that some of them are required to work correctly in the presence of failures of processes. Failures can be either crash failures, or Byzantine (malicious) failures, in which a faulty process can exhibit any behavior.

In this paper, we describe two industrial verification projects of concurrent systems. Both systems are high-level models
of parts of IBM's microcode. These systems model comparatively small fractions of the whole code, but ones that pose most difficulties for verification using standard testing techniques. The first project is verification of the existing protocol that implements page lock mechanism. The second project is a fault-tolerant protocol that we designed and verified. The protocol implements communication between three controllers.

Both protocols were formally verified using the software model checker ExpliSAT and testing tool ConTest. We compare the performance of these tools and discuss their advantages and disadvantages for this type of problems.

We also provide analytical evaluation of the number of control flow paths of deterministic protocols. Based on this evaluation, we compute the minimal number of executions of a system by a random scheduler so that all paths are executed with high probability.

2. PRELIMINARIES

In this section, we briefly describe the verification tools that we used in the case studies.

2.1 ConTest

ConTest [6] belongs to the class of testing tools, noise makers, that make tests more likely to fail and thus increase the efficiency of testing. In the sequential domain, such tools [10] [15] usually work by denying the application certain services, for example, indicating that no more memory is available to a memory allocation request. In the concurrent domain, ConTest force different legal interleavings for each execution of the test in order to check that the test continues to perform correctly. In a sense, it simulates the behavior of other possible schedulers. During the execution of the program, ConTest noise heuristic receives calls embedded by the instrumentor. When such a call is received, the noise heuristic decides, randomly or based on specific statistics or coverage, if some kind of delay is needed.

In addition to increasing the effectiveness of testing by finding races, deadlocks and exceptions early ConTest has a number of other benefits. ConTest enables replay of multi-threaded tests for debugging and regression, supports parallel [2] and sequential coverage models and can be used for simulation of network load through its encapsulation of sockets with noise heuristics.

2.2 ExpliSAT

ExpliSAT [1] is an IBM concurrent software model checker tool. It supports a subset of C++ and the POSIX thread library (pthread). ExpliSAT supports heap memory allocation, dynamic thread creation and unbounded recursion.

The algorithm ExpliSAT uses is a hybrid of traditional explicit-state model checking and SAT-based symbolic model checking. ExpliSAT reduces the number of states it explicitly traverses by exploring only a representative state for each control path. Since several executions may project the same control path, this method reduces the size of the state-space considerably. Yet, since ExpliSAT traverses control flow graph explicitly and since it examines a concrete state and not just a symbolic representation, partial order reductions are as useful as in traditional explicit-state model checking. While explicitly traversing the representative states, ExpliSAT stores all the reachable states symbolically. As a result, ExpliSAT, stores a symbolic representation of all the states reachable during an execution that projects the current control path. The symbolic information is stored as a CNF formula, and a SAT solver is utilized to verify or falsify a property. The SAT solver is also utilized for backtracking only to legal control paths, i.e., control paths that are a projection of a possible execution of the program.

ExpliSAT's implementation uses the C/C++ front-end of CBMC [4] and it employs zChaff [12] as a SAT solver. ExpliSAT supports user-specified assertions and additional implicit properties that must hold in every state, such as "no NULL pointer dereference", "no out-of-bounds array access", "no data race" and "no deadlock".

3. CASE STUDY - PAGE LOCK PROTOCOL

The page manager component of IBM microcode had to be modified due to a new set of requirements. As a result of the change, an abstracted page was implemented using three different pages belonging to three different "generations". Under the original design a multiple-readers-single-writer lock is associated with each page. This semantics had to be maintained when switching to the new design. A new multiple-readers-single-writer protocol was proposed to support the new design but during the design phase the development team was concerned with its validity.

The protocol, partially appearing in Fig 1 in pseudo code, was manually translated to Java and tested with ConTest (see Appendix A). ConTest did not find any problems in the protocol. Then, it was manually translated to C (see Appendix B) and verified using ExpliSAT. In order to ensure the correctness of the protocol, we examine its behavior with respect to 5 different threads that run asynchronously. In order to ensure termination of ExpliSAT, we bound the number of repetitions of the protocol for each thread.

ExpliSAT reports two potential problems of the protocol. The first problem is a possible read-write data-race on the currentLock variable. As can be seen in the pseudo-code, the protocol is aware of such a data-race and handles it by re-checking that the previously read value remains. The second problem reveals the ambiguity of the definition of the critical section in this protocol. In the function changeCurrentLock the critical section ends before the lock is explicitly released. This is against the basic intuition that mutual exclusion is preserved until the lock is released. ExpliSAT verifies that if the access of synchronized variables is restricted to the critical section, the protocol is valid.

Statistics. ExpliSAT completes full analysis of the code in less than a minute when running in Linux OS on a Pentium 4 with 2 GHz and 1GB of memory.

4. CASE STUDY - THREE CONTROLLERS’ PROTOCOL

In this section, we describe modeling and verification of communication protocol between three controllers in IBM microcode. The protocol is part of the code in UNIX Kernel. We received a description of the problem and a set of correctness requirements for the protocol and designed a protocol according to these requirements. The protocol was verified using ConTest and ExpliSAT.
Page Lock Protocol

```
currentLock ∈ {a, b, c}

function: changeCurrentLock(newLock)
{
    // tries to change the current lock
    // to the input lock (the new lock).
    // after trying to lock,
    // it checks whether the local lock
    // is still the same as the global lock.
    // If not, then we have
    // a potential R/W data race.
    // If yes, then the lock
    // was successful, and we are now
    // inside the critical section.
    A: localLock = currentLock;
    lockw(localLock);
    if (localLock ≠ currentLock)
    {
        unlockw(localLock);
        goto A;
    }
    //critical section
    currentLock = newLock;
    //critical section ends here
    unlockw(localLock);
}
```

```
function: accessCurrentPage
{
    A: localLock = currentLock;
    lockr(localLock);
    if (localLock ≠ currentLock)
    {
        unlockr(localLock);
        goto A;
    }
    //do some work
    unlockr(localLock);
}
```

Figure 1: Page Lock protocol

4.1 System model

The system is asynchronous and consists of three controllers with two communication channels (registers) between each pair of controllers (see Figure 2). The controllers communicate by writing data to registers and reading from registers. The communication between each pair of controllers is independent from the communication between other pairs. Therefore, we can focus on one pair of controllers. In each pair there are designated roles of sender and receiver of data: the controller on the left is the sender, and the controller on the right is the receiver. Each controller has a register to which it has read and write permissions. The register of the other controller is read-only. Each register has an indicator bit, which is turned on when there is a new message placed in the register. A controller cannot read the contents of the register if the indicator bit is not turned on. The read action turns the indicator bit off. The communication scheme between two registers is presented graphically in Figure 3. Sender get a message from a client and sends it to the receiver by writing it to \( reg_j \). After successful completion of the protocol, receiver outputs the message, and sender sends an acknowledgment to the client. If the protocol failed, sender sends the failure notification to the client (nack).

The communication protocol between sender and receiver should satisfy the following correctness properties defined over the interface of the system:

1. For each message \( m \), if there exists \( out(m) \), then there also exists \( send(m) \). In other words, each message successfully received by the receiver is a real message sent by a non-faulty sender.

2. For each message \( m \), if there exists \( ack(m) \), then there exists \( out(m) \). In other words, each message successfully sent by the sender is successfully received by a non-faulty receiver.

3. Under the assumptions of partial synchrony, for each message \( m \) such that there exists \( send(m) \), there is eventually \( out(m) \). That is, each sent message is eventually received.

We use the definition of partial synchrony introduced in [5]: there is a global stabilization time of the system, after which there exists an upper bound \( \Delta \) on the completion time of each operation. In the real system, while both controllers are up, this requirement holds. We use time-outs to ensure termination of the protocol in the presence of failures.

4.2 Failure model

The system should withstand failures of controllers. Faulty controllers may exhibit limited Byzantine behavior:

1. When a controller is initially faulty (before it comes up for the first time), its register can contain any value, but it does not change while the controller remains faulty.

2. When a controller was up and crashed, the contents of its register remains the same.

3. The indicator bit of a register that belongs to a faulty controller can have any value at any time.
The system has crash recovery. A controller recovers from failure to the initial state and resets the contents of its register to the value $\bot$.

### 4.3 The protocol

The pseudo-code of the protocol is presented in the figure below. There is no notion of time in our model, thus we implemented time-outs as non-deterministic choices. At each state, each controller can terminate the protocol on time-out and return to the initial state. Since we focus on verifying safety properties, time-outs are omitted from the pseudo-code.

The protocol consists of two communication rounds. In the first round the sender writes $0 \cdot \text{message}$ to $\text{reg}_1$ and waits for an acknowledgment from the receiver. The receiver acknowledges by writing $0 \cdot \text{message}$ to $\text{reg}_2$. In the second round the sender writes $1 \cdot \text{message}$ to $\text{reg}_1$ and the receiver acknowledges by writing $1 \cdot \text{message}$ to $\text{reg}_2$. Both sender and receiver send two different messages each, thus they cannot be faulty for the whole duration of the protocol.

**Sender–Receiver protocol**

**Sender($w$):**

- **state:** \{ initial, wait$_1$, wait$_2$ \}
- **ind$_2$** boolean

**Initialization:**

- **state** = initial
- **$A_i$:** state == initial :
  - write(0,$w$)
  - // $w$ is the message
  - // sender sends to receiver,
  - // 0 is an additional bit
  - **state** = wait$_1$
- **$B_i$:** (state == wait$_1$) \&\& (ind$_2$ == 1):
  - read($rs$)
  - if $rs$ == 0,$w$
  - // first ack received
  - write(1,$w$)
  - **state** = wait$_2$
  - else
  - nack($w$)
  - **state** = initial
- **$C_i$:** (state == wait$_2$) \&\& (ind$_2$ == 1)
  - read($rs$)
  - if $rs$ == 1,$w$
  - // second ack received
  - ack($w$)
  - else
  - Nack($w$)
  - **state** = initial

**Receiver:**

- **state:** \{ initial, wait$_1$, wait$_2$ \}
- **ind$_1$** boolean
- $w_0$, $w_1$ : string
  - // $w_0$ is 0,$w$,
  - // $w_1$ is 1,$w$
- **n** integer

**Initialization:**

- **state** = initial
- **$A_r$:** (state == initial) \&\& (ind$_1$ == 1) :
  - read($w_0$)
  - $n$ = length($w_0$)
  - if ($w_0[0]$ == ‘1’)
  - // a message that looks correct received
  - **write($w_0$)**
  - // first ack is sent
  - **state** = wait$_1$
  - else
  - **state** = initial
  - endif
- **$B_r$:** (state == wait$_1$) \&\& (ind$_1$ == 1)
  - read($w_1$)
  - if ($w_1[0]$ == ‘1’) \&\& ($w_0[1..n-1] = w_1[1..n-1]$)
  - // a message is correct
  - **out($w_1[1..n-1]$)**
  - **write($w_1$)**
  - // second ack is sent
  - **state** = initial

### 4.4 Modeling limited Byzantine failures of controllers

Byzantine failures of controllers are modeled by non-deterministic choices. Since model checking exhaustively checks all possible execution paths, in particular it checks the correctness of the protocol when faulty controllers exhibit the “worst possible behavior”.

In addition to the code that models the normal behavior of the protocol, there are several processes that model failures. The status of both sender and receiver can change non-deterministically at each stage of the execution (from alive to faulty and vice versa). A register of a faulty controller can contain any value, but the value does not change while the controller remains faulty. An indicator bit of this register is changed randomly using random bit generator. In order to separate the code for the normal behavior from the code for faulty behavior, we hide all the access to the input and output from the processes of the sender and the receiver. Instead, the access to the registers is done via special access functions that also take into account the state of the controller (normal or faulty). The result is that the code of the normal behavior can be used by the developers as a basis for implementation.

The protocol was written in Java for testing with ConTest (see Appendix C) and in C for verification with ExpliSAT (see Appendix D).

The resulting implementation in C is based on the pseudo-code in Section 4.3 and it consists of 400 lines of code. It is a straightforward implementation of the pseudo-code.

### 4.5 Verification with ConTest

The Java code that is the input to ConTest separates the threads for normal behavior of controllers (NormalSender and NormalReceiver) from the threads that emulate failed controllers (FaultySender and FaultyReceiver). We follow this approach, of maximal separation between the code for the normal behavior and the code that emulates failures, in both Java and C code.

The model was verified in ConTest. The run took several minutes and was terminated manually when all synchronization primitives of the Java code got 100% coverage, that is, were visited in three modes: blocking, blocked, and passing (neither blocking, nor blocked). This coverage metric, of checking the modes in which synchronization primitives are visited, clearly cannot guarantee that all possible paths were checked. It only checks that all synchronization primitives were visited in all possible modes. However, it is usually
a good heuristic measure of exhaustiveness of the testing procedure.

Since ConTest uses randomness, in the general case there is no guarantee that all possible behaviors of the system are covered during a finite run. In this case the model is sufficiently small to allow manual examination of exhaustive check of all possible behaviors. In Section 6 we introduce an estimate on the number of behaviors in deterministic concurrent protocols, and in Section 7 we estimate the number of times ConTest should execute the system in order to assert with a high degree of confidence that all behaviors were checked.

4.6 Verification with ExpliSAT

As opposed to ConTest, ExpliSAT deterministically checks all possible behaviors of the program. Thus, at the end of the run of ExpliSAT on the program we are guaranteed to have examined all behaviors. The model of the three controllers’ problem for ExpliSAT differs from the model for ConTest mainly in modeling of timeouts. Since ExpliSAT is a static analysis tool, and the model does not contain time, we cannot measure time-outs. Instead, a controller can terminate the protocol non-deterministically on time-out at each state where time-outs are allowed. The following lemma state that this modeling is precise.

Lemma 4.1. All behaviors with time-outs that exist in the original system are preserved with modeling time-outs as non-deterministic decisions, and this modeling does not create spurious counter-examples.

Indeed, it is easy to see that for each execution path that includes time-out, there exists a matching execution path in our model, where time-out is replaced by non-deterministic decision to interrupt the protocol. For the other direction, note that every non-deterministic exit from the protocol where time-out is allowed can occur with sufficiently slow responses from the other party in the protocol.

Statistics. As with the page lock protocol, ExpliSAT completes full analysis of the code in less than a minute when running in Linux OS on a Pentium 4 with 2 GHz and 1GB of memory.

4.7 Duplicate messages

In this section, we discuss the scenario in which a message is successfully received by the receiver, yet is marked as undelivered by the sender. The scenario is as follows:

1. Sender sends 0 \cdot message.
2. Receiver acknowledges.
3. Sender sends 1 \cdot message.
4. Receiver marks the transaction as successfully completed, outputs message and acknowledges to the sender.
5. Before receiving the second acknowledgment, sender exits on time-out, aborting the transaction and returning nack on message.

As a result, the same message can be successfully received by a client on the receiver side and marked as not delivered by a client on the sender side. Thus, number of successfully received messages can be greater than the number of successfully sent messages. Note that this scenario does not falsify any of the specifications we checked in this case study. Moreover, this problem is unavoidable due to time-outs and does not depend on the number of communication rounds. Indeed, for any number of communication rounds, the scenario above can occur at the end of the last round.

In the protocol, each message is treated separately, so there is no information about whether the message was already sent. Depending on the application, the scenario in which the message is successfully received but not acknowledged may or may not be a problem. An application may choose to mark the message as undelivered or to try to resend it. In this case, the same message can be delivered more than once. Depending on the nature of the message, this might be something an application wants to avoid. As an example of harmless duplicate messages, consider the situation in which sender informs the receiver of the status of the system by sending a message. Receiving the same status message more than once is clearly harmless. On the other hand, consider a situation in which the receiver is a printer, and a sender sends a large job divided into blocks. In this case, duplicate messages result in printing a job in which some blocks appear more than once, so duplicate messages should be identified.

An application can identify duplicate messages by keeping bounded counters. A counter modulo \( k \) can be attached to each message. In this case, a message that is not acknowledged is resent with the same counter. The client on the receiver side can store the last value of the counter it saw and compare it with the counter of the next message. If the values are the same, the message was already received and can be discarded (depending on the semantics of the application). Two-bit counters suffice, yet one-bit counter is not enough, as shown in the following scenario:

1. Sender sends a message \( m_1 \) with counter 0.
2. Receiver receives and acknowledges.
3. Sender sends a message \( m_2 \) with counter 1.
4. Receiver receives, acknowledges, and fails before it saves the current value of the counter.
5. Receiver comes up.
6. Sender sends a message \( m_3 \) with counter 0.
7. Since the last counter receiver recorded is 0, it assumes that sender re-sends the message \( m_1 \), so it discards the message \( m_3 \).
8. The result: message \( m_3 \) is lost.

In order to check the application with counters, we modify the protocol so that the sender sends messages from an array of messages. If a message is not sent successfully to the receiver, sender resends it with the same counter until it is successfully delivered. The C code in Appendix D implements the protocol with counter modulo 4.

5. COMPARISON BETWEEN CONTEST AND EXPLISAT

One of the most important features of verification tools is their ability to check programs written in the language used
by the developer. In this case, developers use C as their primary programming language. Since the case studies are quite small, it is possible in theory to translate them manually into HDL and verify with one of the existing model checkers (for example, RuleBase of IBM). This, however, introduces an additional error factor that stems from the manual translation. Moreover, the verified program in HDL cannot be presented to the developer as is, and a translation in the other direction is needed. Therefore, use of a specialized software verification tool is essential for both case studies. While ExpliSAT is a model checker tailored especially for C and C++, ConTest works with Java. The translation from C to Java and back, however, does not pose a substantial difficulty.

The programs in both case studies are quite small. ConTest was manually terminated after several minutes on each program, and ExpliSAT terminated after less than a minute having performed complete verification. We note that, strictly speaking, ConTest does not terminate. On each run it chooses a possible scheduling randomly, without keeping record of the previous runs. The progress of ConTest can be heuristically measured by using coverage metrics, which are widely used in industry as an indication of exhaustiveness of testing tools. The coverage metric ConTest uses counts visits to synchronization primitives, and is a somewhat restricted type of code coverage metric. ExpliSAT, on the other hand, deterministically explores all possible execution paths and terminates when all paths of the program were checked. However, for non-terminating programs ExpliSAT also fails to terminate, albeit not from the same reason as ConTest. The reason is that non-terminating programs create a possibly infinite number of behaviors. ExpliSAT, as of now, does not recognize that the number of behaviors is in fact finite even in presence of infinite loops.

ConTest does not need any adjustment of the program, except for the removal of time-outs. Since ConTest introduces time-outs of its own, any existing time-outs in the program are changed or ignored. ExpliSAT requires a program to be written in a special way due to the limitations on the programming language it currently understands. Also, the correctness requirements have to be inserted into the program in the form of assertions.

To summarize, we feel that ConTest is perhaps a little easier to work with than ExpliSAT. On the other hand, the lack of exhaustiveness of ConTest allows us to verify a program only with a certain degree of confidence in its correctness, whereas ExpliSAT, being a model checker, guarantees correctness of the program in case of successful termination. Also, as we discuss below, ConTest is more likely to execute short paths, due to its randomized selection of scheduling. As a result, longer paths may not be executed at all. In what follows, we discuss ways to tune ConTest so that its exhaustiveness can be measured precisely, and the probability to execute a path is the same for all paths. This feature is currently not implemented in ConTest and is left for future work.

6. ESTIMATION OF THE NUMBER OF CONTROL FLOW PATHS IN DETERMINISTIC PROTOCOLS

In this section we examine the number of control flow paths of processes in deterministic memory-less protocols

\[ \pi \]

without recovery. Deterministic nature of the protocol means that there is exactly one control flow path in case all participating processes are not faulty and no time-outs or other failures occur. The protocol is memory-less if every subsequent iteration of the protocol does not depend on the previous iterations. The “no recovery” characteristic means that in case of a failure of the protocol, there is no way to recover, and the processes return to the initial state.

The two protocols we examine in this paper have all these characteristics. The modification of the second protocol where a bounded counter is introduced is not memory-less, but the original protocol, without counters, is memory-less\(^1\). Both protocols assume that in case of a failure the processes return to the initial state. The control flow of such a protocol is presented schematically in Figure 4, where back edges to the initial state are omitted. At each state, at most three actions are enabled: continuing the correct execution, exiting on timeout or faulty behavior of another process, and failure. In general, for a process with \(k\) states, the number of transitions is bounded by \(k^2\), and the number of different control flow paths is exponential in \(k^2\). For two processes running concurrently, each with \(k\) states, the number of states in the concurrent program is exponential in \(k^3\) (all possible pairs of states), and thus the number of transitions is bounded by \(k^4\), and the number of different control flow paths is exponential in \(k^5\), that is, quadratic in the number of control flow paths of a one process.

![Figure 4: Control flow of one process in a a deterministic protocol](image)

In a special case of bounded branching degree and termination of the protocol in a case of failure, as in the protocols we examined, we are able to achieve a better bound. First, we estimate the number of control flow paths of one process executing the protocol. Let \(m\) be the number of states in the normal path, and let \(s\) be the number of sink (failure) states. Then, there is one path for the normal behavior, and \(m\) possibilities to interrupt the normal behavior, which lead to one of \(s\) sinks. In total, we have \((1 + s \times m)\) paths. For a system of two processes running concurrently without synchronization, the number of control flow paths of the whole system is exponential in \(m\). In the three controllers’ protocol, however, the processes are synchronized in almost every state. Thus, for a pair of control flow paths \((\pi_1, \pi_2)\), where \(\pi_1\) is a control flow path of the sender and \(\pi_2\) is a control flow path of the receiver, there exists at most one legal control flow path \(\pi\) of the whole system in which the sender follows \(\pi_1\) and the receiver follows \(\pi_2\). Therefore, the number of

\(^1\)If using bounded counters, a similar claim is true for each \(k\) subsequent repetitions of the protocol, where \(k\) is the bound on the counter.
control flow paths of a system with two highly synchronized processes is $O(s^2m^2)$, which is clearly much less than an exponential bound we have in the general case. Note that we abstract the data values away, so in order to estimate the number of paths including all possible data values we have to multiply this number by the number of combinations of data values. Since not all values are possible, this estimation is an upper bound.

7. ESTIMATION OF THE RUNNING TIME OF RANDOM SCHEDULERS

In this section, we study the problem of estimating the number of times a random scheduler has to execute a program in order to execute all its paths with high probability. A random scheduler is a scheduler that at each state of the program that allows several choices of the next state, chooses each successor with equal probability. ConTest has a random scheduler, so this discussion is especially relevant for ConTest. We start with a simple setting in which the control flow graph of a program is a full tree, and then extend our conclusions to general control flow graphs.

7.1 A simple setting

We assume that the control flow graph of a program is a full tree with branching degree $d$ and height $m$. That is, all control flow paths are of the same length, and since each one of them has the same number of states with multiple successors and all branching degrees are equal, each path has the same probability to be chosen in the execution. Let $n = dm$ denote the number of paths in the tree.

Under this setting, the problem of estimating the number of times a scheduler executes the program before all paths are covered is similar to the “coupon collector’s problem” (see [11] for a detailed study of this problem). After we have seen $i − 1$ paths, the probability $p_i$ of seeing a different path in the next run is $p_i = 1 − \frac{1}{n}$. Let $X$ be the random variable of the number of executions before we see all paths. Then, the expected number of executions $E(X)$ before all paths are seen is $E(X) = n \times H(n)$, where $H(n)$ is the harmonic number $\sum_{i=1}^{n} 1/i$, which is equal to $\ln n + \Theta(1)$. Therefore, the expected number of executions to cover all $n$ paths is

$$E(X) = n \ln n + \Theta(n),$$

that is, greater than the number of paths by the multiplicative factor of $\ln n$.

In many cases we are not interested in the expected number of executions, but rather in the number of executions that will cover all paths with high probability. Given a probability $0 < p < 1$, we want to calculate the number of times the scheduler should run the program before it sees all $n$ its paths with probability at least $p$. Similarly to the coupon collector’s problem, the probability of not seeing an $i$th path after $n \ln n + cn$ executions is

$$\left(1 - \frac{1}{n}\right)^{\ln n + c} < e^{-c} = e^{\ln e^{-c}} = \frac{1}{e^c}.\]

By a union bound, the probability that we have not seen some path after $n \ln n + cn$ executions is bounded by $e^{-c}$. Since we require this probability to be less than $1 - p$, we have $e^{-c} < 1 - p$, which holds for all $c > -\ln (1 - p)$. In particular, for $p = 2/3$, it is sufficient to choose $c = 1.1$. In practice, we can use this estimation in conjunction with the upper bound on the number of executions from Section 6 to compute the number of times we have to execute the scheduler in order to achieve full coverage of paths of a program with high probability.

7.2 General graphs

In general graphs, paths can be of different length. For example, in the control flow graph in Figure 4 the longest path that stands for the normal completion of the protocol is of length $m$, while other paths are much shorter. If each successor is chosen with the same probability, the probability of choosing the longest path is $1/dm$. Thus, the expected number of executions of a program to cover the longest path is $O(dm)$, which is exponentially larger than the number of paths in this graph ($O(d \times m)$). The computation in Section 7.1 can be viewed as an approximation of the estimation of the required number of execution for a general graph, when the graph is unwound to a tree and complemented to the full tree by adding “dummy paths”. If we unwind the graph in Figure 4 and complement it to the full tree, the number of paths in the resulting tree is $d^m$, and thus the expected number of executions computed as in Section 7.1 is $d^m \ln d^m = O(d^m m)$. In other words, the approximation obtained by unwinding is very close to the actual number of times we have to execute a random scheduler.

Since the number of paths in a general graph can be far from exponential, an exponential number of executions is too high. We need a way to increase the probability of visiting long paths, so that the expected number of executions of a program by a random scheduler will be polynomial in the actual number of paths. We present a solution to this problem for DAGs (directed acyclic graphs). Note that we are interested in single executions. That is, if a program runs continuously but it is memory-less, the graph of a single execution of it does not have back-edges to the initial state (see, for example, Figure 4).

**Theorem 7.1.** Let $G$ be a control flow graph of a program with the initial state $s_0$. For a state $s$, let $P(s)$ be the number of paths of $G$ that originate in $s$. For each state $s$ and its successors $s_1, \ldots, s_d$ of $s$, we set the probability of a random scheduler to choose the successor $s_i$ when in state $s$ to $P(s_i)/\sum_{i=1}^{d} P(s_i)$, for $1 \leq i \leq d$. If $G$ is a DAG, then each path in $G$ is executed with equal probability.

The theorem is proved by induction on the structure of the graph. They key observation is that the number of paths originating in $s$ is equal to $\sum_{i=1}^{d} P(s_i)$.

If a random scheduler chooses paths as in Theorem 7.1, the conditions of the coupon collector problem hold for the actual number of paths in the system. Then, for deterministic protocols as in Section 6 the expected number of executions of a system in order to reach all paths is $O(dm \ln dm)$ for a single process, where $d$ is the branching degree ($d = s + 1$), and $m$ is the number of states in the normal paths. If the system has many processes running concurrently, the number of paths increases, and thus the expected running time of the scheduler increases as well. However, it is always equal to the actual number of paths in the system multiplied by a logarithmic factor.

8. CONCLUSIONS

Both protocols we discuss in this paper are described in a high level of abstraction. This allowed a quick and effi-
cient verification process, and errors found during verification were easy to understand. At the same time, both protocols model the behavior of concurrent system and were very hard to check using traditional testing techniques. So, the benefit in using formal verification methods here is clear.

In both cases, the protocols represent a relatively small sections of the whole system. These are, however, the sections with the highest complexity, and thus, the highest likelihood to contain errors. The page lock protocol is a part of the code that contains about 3000 lines, while the protocol itself contains about 300 lines. However, the remaining 2700 lines of code are straightforward and do not have any complex behavior. Thus, verifying 1/10 of the total code was enough to ensure correctness of the whole system. The part of the code that implements the three controllers’ protocol is 400 lines, whereas the whole system consists of thousands of lines of code. At the same time, the problem of ensuring a correct communication between the controllers was the main obstacle to verifying the correctness of the system.

We compared ConTest and ExpliSAT on both protocols. While efficiency is not an issue since the programs are so small, other criteria such as ease of usage, guarantee of exhaustiveness, and comprehensiveness of the results are still relevant. In both protocols, ConTest overlooked several problems discovered by ExpliSAT. Also, as we showed in Section 7, running ConTest without adjusting the probabilities it attributes to non-deterministic choices leads to exponential number of executions in order to execute all paths with high probability. In any case, since ConTest is a probabilistic testing tool, it can achieve exhaustiveness only with high probability. As opposed to ConTest, ExpliSAT guarantees exhaustiveness. It also does not need to be tuned to the structure of the control flow graph, since it explores all paths deterministically. However, using it is harder than using ConTest, both because of the special style of encoding the protocol (see, for example, the use of assume in the code), and because its output is not as user-friendly as the output of ConTest. We are currently working on improving the working environment of ExpliSAT and making it available to more users inside IBM.

9. ACKNOWLEDGMENTS

Authors thank Irit Cohen for describing the page lock protocol and implementing it in the IBM microcode after it was formally verified.

10. REFERENCES


APPENDIX

A. JAVA CODE FOR PAGE-LOCK PROTOCOL

```java
public static void main(String arg[]){
    //TBC - make the table changes random
    LOCKS.currentLock = LOCKS.a;
    if(debug == true)
        System.out.println
        ("LOCKS.currentLock = " + LOCKS.currentLock);
    PageLock changeTable = new PageLock(Locks.a);
    Thread changeTableThread = new Thread(changeTable);
    if(debug == true)
        System.out.println
        ("change process to lock " + LOCKS.a + " started");
    PageLock changeTableThread = new PageLock(Locks.a);
    changeTableThread.start();
    PageLock changeTableThread = new PageLock(Locks.b);
    Thread changeTableThread1 = new Thread(changeTable);
    if(debug == true)
        System.out.println
        ("change process to lock " + LOCKS.b + " started");
    changeTableThread1.start();
    PageLock changeTableThread2 = new PageLock(Locks.c);
    Thread changeTableThread2 = new Thread(changeTable2);
    if(debug == true)
        System.out.println
        ("change process to lock " + LOCKS.c + " started");
    changeTableThread2.start();
```

18
System.out.println
("change process to lock = LOCKS.c " + started");
changeTableThread2.start();
}

public void run(){

System.out.println
(Thread.currentThread() + "started");
while(true){
   while(i <= 2){
      g = LOCKS.currentLock;
      if(debug == true)
         System.out.println
         ("thread - " + Thread.currentThread() +
         "local g lock reference set to " + g);
      synchronized(g){
         if(debug == true)
            System.out.println
            ("thread - " + Thread.currentThread() +
            "lock - " + g + " was obtained");
         if(g != LOCKS.currentLock){
            if(debug == true)
               System.out.println
               ("thread - " + Thread.currentThread() +
               "giving up as race occurredlocal lock is -" + g + "while global lock is " +
               LOCKS.currentLock);
            } else {
               synchronized(h){
                  if(debug == true)
                     System.out.println
                     ("thread - " + Thread.currentThread() +
                     "obtained second lock - " + h);
                  if(settingF == true)
                     System.out.println("protocol failed");
                } else {
                   settingF = true;
                   LOCKS.currentLock = h;
                   settingF = false;
                   if(debug == true)
                      System.out.println
                      ("thread - " + Thread.currentThread() +
                      "updated current lock to -" + h);
                   i++;
               }
           }
       }
   }
}

class LOCKS {
static Object a = new Object();
static Object b = new Object();
static Object c = new Object();
static Object currentLock;
}

B. C CODE FOR PAGE-LOCK PROTOCOL

void changePageLock(lock *newLock) {
    lock *localLock = globalLock;
    lockGetWriteLock(localLock);
    if (localLock != globalLock) {
        lockReleaseWriteLock(localLock);
        continue;
    }
    // critical section starts here
    x++;
    assert (x<2); //verify mutual exclusion
    x--
    globalLock = newLock; //critical section ends here
    x++;
    assert (x<2); // mutual exclusion here was falsified
    x--;
    //critical section might appear as ending here
    lockReleaseWriteLock(localLock);
}

void currentPageLock() {
    lock *localLock = globalLock;
    while (true) {
        lockGetReadLock(localLock);
        if (localLock != globalLock) {
            lockReleaseReadLock(localLock);
            continue;
        }
        //critical section here
        x++;
        assert (x<2); // verify mutual exclusion
        x--;
        //critical section ends here
        lockReleaseReadLock(localLock);
    }
}

void * employProtocol(void *foo) {
    int i;
    for (i=0; i<N; i++)
    {
        // nondeterministically decide what to employ
        switch (nondet_int()){
            case 0:
                changeCurrentPage(lockA);
                break;
            case 1:
                changeCurrentPage(lockB);
                break;
            case 2:
                changeCurrentPage(lockC);
                break;
            case 3:
                currentPageLock(x);
                break;
            default: // do nothing
                break;
        }
    }
}

main () {
    pthread_t trd;
    lockInit(lockA);
    lockInit(lockB);
}
C. JAVA CODE FOR THREE CONTROLLERS’ PROTOCOL

private static class NormalSender extends Thread {
    private void round(Msg msg) throws ProtocolFailure {
        synchronized (sendersRegister) {
            sendersRegister.setMsg(msg);
            sendersRegister.setReceived(true);
        }
        synchronized (receiversRegister) {
            receiversRegister.waitForReceived();
            receiverMessage = receiversRegister.getMessage();
            roundNumber = receiversRegister.getRoundNumber();
            receiversRegister.setReceived(false);
        }
        if (receiverMessage != msg.message 
            || roundNumber != msg.roundNumber) {
            throw new ProtocolFailure
                ("receiver is faulty, terminated transaction");
        }
    }

    public void run() {
        while (true) {
            senderBeginningOfCycle = true;
            senderBeginningOfCycle = false;
            state = ControllerState.INITIAL;
            msg.message = toss();
            msg.roundNumber = 0;
            state = ControllerState.WAIT1;
            round(msg);
            msg.roundNumber = 1;
            state = ControllerState.WAIT2;
            round(msg);
            if (sendersStatus.isAlive()) {
                System.out.println
                    ("transaction succeeded, sent " + msg.message);
            }
        }
    }
}

private static class FaultySender extends Thread {
    public void run() {
        while (true) {
            if (! sendersStatus.isAlive()) {
                sendersRegister.setReceived(QuickRandom.nextBoolean());
            }
        }
    }
}

private static class ReceivedBitRandomizer extends Thread {
    public void run() {
        while (true) {
            if (! sendersStatus.isAlive()) {
                sendersRegister.setReceived(QuickRandom.nextBoolean());
            }
        }
    }
}

public static void main(String[] args) {
    new ReceivedBitRandomizer().start();
    new NormalReceiver().start();
    new NormalSender().start();
}
D. C CODE FOR THREE CONTROLLERS’ PROTOCOL

typedef struct {
    int roundNumber; //0 or 1
    int count;      // 0..4
    int data;       //in this example, 0 or 1
} Msg;

void next_msg_num(int* count)
{ *count = (*count +1) % 4; }

void Sender_round(Msg* pmsg, bool* failed) {
    Msg receivedMsg; bool received;
    putMsg(&sendersRegister, pmsg);
    received = waitWithTimeout(&receiversRegister, TIMEOUT);
    if( received )
        receivedMsg = readMsg(&receiversRegister);
    if( (!received || receivedMsg.data != pmsg->data ||
         receivedMsg.roundNumber != pmsg->roundNumber) )
        *failed = true; // timeout or faulty receiver
}

void Sender_run() {
    ControllerState state;
    bool failed, new_message;
    int count = 0;
    int msg0;
    while (sentMsgs < NUM_MSGS) {
        state = INITIAL;
        new_message = Receiver_round(0,count,&failed,&msg0);
        if( failed ) continue;
        state = WAIT1;
        new_message = Receiver_round(1,count,&failed,&msg0);
        if( failed ) continue;
        if (new_message) {
            receivedQueue[receivedMsgs++] = msg0;
            next_msg_num(&count);
            if(receivedMsgs > 0)
                assert( sendQueue[receivedMsgs-1] ==
                        receivedQueue[receivedMsgs-1] );
            assert( receivedMsgs < sentMsgs+2 );
        }
    } /* while */
}

bool Receiver_round(int roundNum, int count,
    bool* failed, int* msg0)
{ // returns true when receives new message.
    bool received, roundOK, countOK, msgOK;
    received = waitWithTimeout(&sendersRegister,TIMEOUT);
    if( received ) {
        receivedMsg = readMsg(&sendersRegister);
        roundOK = (receivedMsg.roundNumber == roundNum);
        countOK = (receivedMsg.count == count) ||
                  ((receivedMsg.count+1) % 4 == count);
        msgOK = (roundNum == 0) || (*msg0 == receivedMsg.data);
        if( received & roundOK & countOK & msgOK )
            putMsg(&receiversRegister, &receivedMsg);
        else
            *failed = true;
        if( roundNum == 0 )
            *msg0 = receivedMsg.data;
        return(!*failed & receivedMsg.count == count);
    }

    void Receiver_run() { /*
        ControllerState state;
        int count = 0;
        bool failed, new_message;
        int msg0;
        while (receivedMsgs < NUM_MSGS) {
            state = INITIAL;
            new_message = Receiver_round(0,count,&failed,&msg0);
            if( failed ) continue;
            state = WAIT1;
            new_message = Receiver_round(1,count,&failed,&msg0);
            if( failed ) continue;
            if (new_message) {
                receivedQueue[receivedMsgs++] = msg0;
                next_msg_num(&count);
                if(receivedMsgs > 0)
                    assert( sendQueue[receivedMsgs-1] ==
                            receivedQueue[receivedMsgs-1] );
                assert( receivedMsgs < sentMsgs+2 );
            } /* while */
        }
    }