Efficient Model Checking of Applications with Input/Output

Cyrille Artho[1], Boris Zweimüller[2], Armin Biere[3], Etsuya Shibayama[1], and Shinichi Honiden[4]

[1] Research Center for Information Security (RCIS),
National Inst. of Advanced Industrial Science and Technology (AIST), Tokyo, Japan
[2] Computer Systems Institute, ETH Zürich, Switzerland
[3] Johannes Kepler University, Linz, Austria
[4] National Institute of Informatics, Tokyo, Japan

Abstract. Most non-trivial applications use some form of input/output (I/O), such as network communication. When model checking such an application, a simple state space exploration scheme is not applicable, as the process being model checked would replay I/O operations when revisiting a given state. Thus software model checking needs to encapsulate such operations in a caching layer that is capable of hiding redundant executions of I/O operations from the environment.

Keywords: Software model checking, network communication, software testing

1 Introduction

Model checking explores the entire behavior of a system under test (SUT) by investigating each reachable system state [5] for different thread schedules. Recently, model checking has been applied directly to software [2,4,6,7,13]. However, conventional software model checking techniques are not applicable to networked programs. The problem is that state space exploration involves backtracking. After backtracking, the model checker will execute certain parts of the program (and thus certain input/output operations) again. However, external processes, which are not under the control of the model checking engine, cannot be kept in synchronization with backtracking, causing direct communication between the SUT and external processes to fail.

Our work proposes a solution to this problem. It covers all input/output (I/O) operations on streams and is applicable as long as I/O operations of the SUT always produce the same data stream, regardless of the non-determinism of the schedule.

This paper is organized as follows: An intuition for our algorithm is given in Section 2, while Section 3 formalizes our algorithm. Experiments are given in Section 4. Section 5 describes related work. Future work is outlined in Section 6, which concludes this paper.

2 Intuition of the Caching Algorithm

Model checking of a multi-threaded program analyzes all non-deterministic decisions in a program. Non-determinism includes all possible interleavings between threads that can be generated by the thread scheduler. Alternative schedules are explored by storing the current program state and executing copies of said program state under different schedules. When model checking a SUT that is part of a distributed system using multiple processes, external processes are not backtracked during model checking. Thus, two problems arise:

1. The SUT will re-send data after backtracking. This will interfere with the correct functionality of an external process.
2. After backtracking, the SUT will expect external input again. However, an external process does not re-send previously transmitted data.

One possible solution to this problem is to lift the power of a model checker from process level to operating system (OS) level. This way, any I/O operation is under control of the model checker [10]. However, this approach suffers from scalability problems, as the combination of multiple processes yields a very large state space.

Similar scalability problems arise if one transforms several processes into a single process by a technique called *centralization* [11]. With a model for TCP/IP, networked applications can be model checked, but the approach does not scale to large systems [1,3].

Our approach differs in that it only executes a single process inside the model checker, and runs all the other applications externally. Inter-process communication is supported by intercepting any network traffic in a special cache layer. This cache layer represents the state of communication between the SUT and external processes at different points in time. After backtracking to an earlier program state, data previously received by the SUT is replayed by the cache when requested again. Data previously sent by the SUT is not sent again over the network; instead, it is compared to the data contained in the cache. The underlying assumption is that communication between processes has to be independent of the thread schedule. Therefore, the order in which I/O operations occur must be consistent for all possible thread interleavings. If this were not the case, behavior of the communication resource would be undefined. Whenever communication proceeds beyond previously cached information, the new data is both physically transmitted over the network and also added to the cache. The only exception to this is closing a connection. The cache simulates the effect of closing communication but allows connections to remain physically open for subsequent backtracking.

For this approach to work, communication with the environment must be independent of thread scheduling. Therefore, the order in which I/O operations occur and data is sent, must be consistent for all possible thread interleavings. If this were not the case, behavior of the communication resource would be undefined, as all communication is assumed to be deterministic.

3 Formalization of Stream I/O with Rollback

The following definitions assume a semantics for variables as in computer programs, allowing for updates of variables and functions using assignment operator $:=$.

3.1 Stream abstraction

Programs operate on a set S of data streams (which correspond to streams or sockets in a given programming language). A communication trace t is a finite sequence of data: $t = \langle t_0, \ldots, t_i \rangle$. Without loss of generality, we assume a uniform size for all data values used. Let $|t|$ denote the length of a trace and T the set of all traces. A data stream $s\,\langle t, st \rangle$ consists of a communication trace t and a stream state st. A stream state $st\,\langle c, p \rangle$ consists of a connection state c where $c \in \{open, closed\}$, and the current position p in the associated communication trace. Function $\mathrm{t}(s)$ returns the corresponding trace of a data stream; $c(s)$ and $p(s)$ return the connection state of a stream state and its position, respectively. The current stream state for a given stream s is returned by function $\mathrm{state} : s \rightarrow st$.

A milestone m consists of the full program state, including current stream states. Stored stream states are modeled by function mstate. This function returns a function containing stored stream states, $\mathrm{mstate} : m \rightarrow \mathrm{state}$. Let "linear mode" denote execution when no milestone is active, and "milestone mode" an execution trace during which at least one milestone has been created (and not yet removed). In milestone mode, the model checker maintains a set of all milestones M.

3.2 Execution semantics

Model checking of a program using I/O is performed as follows: In linear mode, all operations are directly executed, using the functionality provided by the standard library. The result of the library function, using the correct set of parameters, will be denoted by `lib`$(\ldots)$. In milestone mode, all subsequent changes to communication traces are recorded in T. Communication traces in T are recorded globally, outside each milestone.

A communication trace of a given stream, $t(s)$, is *consistent* w.r.t. a previously seen communication trace $t \in T$ iff, for the current schedule, the same trace is encountered up to position $p(s)$ of stream s. In our approach, all communication traces have to be consistent with the first seen communication trace. During any non-deterministic run of a program, there has to be one unique communication trace t' such that for all thread schedules, $t(s) = t'$.

All data sent over a stream has to be equal across all possible program schedules, as the schedule should not change application behavior. Actions extending the system state beyond a previously cached state extend cache information with new data. In order to denote this, position p reflects the current position of the

cached trace, which is increased when the current trace exceeds a previously cached one, as defined below.

Creation of a milestone m requires the model checker to record the current state of all streams in m, i.e., $\text{mstate}(m) := \mathit{state}$. In milestone mode, *execution* behaves as follows:

- Reading data: $\texttt{read}(s)$ returns $\begin{cases} t_{p(s)}(s) \text{ if } p(s) \leq |t(s)| \\ \texttt{lib}(\ldots) \text{ otherwise} \end{cases}$
  This operation also sets $t_{p(s)}$ to the value returned and increments $p(s)$ after that.
- Writing data: $\texttt{write}(s, d)$ $\begin{cases} \text{checks if } t_{p(s)}(s) = d \text{ if } p(s) \leq |t(s)| \\ \text{calls } \texttt{lib}(\ldots) \text{ and sets } t_{p(s)} := d \text{ otherwise} \end{cases}$
  If $t_{p(s)} \neq d$, the program trace is inconsistent with a previously checked schedule, and model checking is aborted. Otherwise, $p(s)$ is incremented after access to $t_{p(s)}$.
- Opening a stream: $\texttt{open}(s)$ $\begin{cases} \text{returns an error if } c(s) = \mathit{open} \\ \text{sets } c(s) := \mathit{open} \text{ otherwise} \end{cases}$
- Closing a stream: $\texttt{close}(s)$ $\begin{cases} \text{returns an error if } c(s) = \mathit{closed} \\ \text{sets } c(s) := \mathit{closed} \text{ otherwise} \end{cases}$
  The error codes returned for `open` and `close` correspond to the ones returned by `lib` in the same situation.

A *rollback* operation affects the state of each stream in m, restoring them to their previous value: $\text{state} := \text{mstate}(m)$. The cached communication trace t of each data stream is not reverted. If previously recorded parts of a communication trace are re-sent by the model checker, they are only accepted if they match the given history. Changes in t reflect the fact that the current exploration sent more data to the network than in previously seen subsets of the state space.

When a milestone is *removed* from M, its associated state information is discarded. Communication traces are stored and mapped as long as milestones containing them exist. A stream is physically closed when the last milestone containing it is removed and $c(s) = \mathit{closed}$.

3.3 Limitations of replay-based approaches

Any program whose communication fulfills the criteria defined above can be model checked successfully using our approach. However, there are classes of programs that are normally considered to be valid, for which our criteria are too strict. This includes software that logs events to a file or network connection. For this discussion it is assumed that logging occurs by using methods `open`, `write`, and `close`. Assume further that actions of each thread can be interleaved with actions of other threads, which include logging.

If log entries of individual threads depend on thread-local data, they are independent of each other. In such a case, different correct interleavings of log entries can occur without violating program correctness. If log data is sent over a single shared communication channel, occurrence of different message interleavings violates the criterion saying that written data at a specific position must

be equal for all thread interleavings. Such programs can therefore not be model checked with our approach, unless some messages were treated specially, e.g. by ignoring the order in which they appear in the trace.

Note that this limitation only applies if several threads share the same connection. If each thread has its own connection, then the order in which connections are used may be affected by the schedule, but as long as the content of each communication trace does not vary across schedules, our consistency criterion is fulfilled.

On a more general level, applications where communication depends on the global application state are not applicable to our approach. A chat server that responds to requests by sending the number of currently connected clients is an example for this. Communication (the server response) depends on the total number of clients connected. Assume a chat *client* is run inside the model checker, which has two threads connecting to the server. When replaying a partial communication trace of one thread, this communication trace may not match with a previously seen trace, because the number of clients currently connected varies depending on the thread schedule. Similar problems appear if communication content depends on the state of other processes. Such cases can only be model checked by using application centralization [1,3].

3.4 Limitations of our implementation

Our approach is strictly stream-based. Our initial implementation does not properly distinguish between communication channels used by different threads, and therefore does not work on more complex applications. While we have successfully run our tool on a simple web server, where each request and response consists of a single, atomic message, the implementation fails for more complex protocols. Work is in progress to address this problem.

4 Experiments

It is not obvious how communicating applications can or should be tested. Clearly, it is necessary to have at least two communicating applications: the SUT running inside a model checker, and the remote application running independently. Even though applications can be truly distributed, running on different hosts, it is necessary to execute them on the same host for expedient automation of such a test. In order to allow for easier automation, both the internal and "remote" application are launched by our model checker, JNuke.

Both the SUT and the remote application have to be synchronized for initiating a test. If this was not the case, it could happen that a client attempts to contact a server that is not ready yet. Indeed, it is not trivial to avoid such a scenario, as the state of the external application cannot be supervised or influenced by the model checker. There are two ways to prevent premature client startup:

1. Extra control code could be added to the client, ensuring that the server is ready. For instance, the client could retry a communication attempt in the event of failure.
2. Starting the client is delayed after starting the server. This allows the server to initialize itself and be ready for communication.

The second approach is less reliable, but more practical, as it does not require modification of the SUT. Experiments with this approach worked quite reliably under different settings. Reliability could be further improved by using operating system utilities to supervise system calls. Such tools include `trace`, `strace`, and `truss` [8]. Unfortunately, it is not possible to access them in a uniform manner from inside an application on different platforms.

Initial experiments tested the performance of our I/O layer with no model checking enabled (linear execution). These tests were run on two Pentium 4 computers with a clock frequency of 1.3 GHz and 512 MB of RAM. Table 1 shows that our implementation is not yet quite as fast as the one in Sun's VM but delivers a comparable performance for sending larger bursts of data (more than 1 KB per call). For experiments, 100 MB were transmitted. User time (the time spent while executing bytecode and native methods in the VM) and total time (real time) were measured. The difference indicates the time needed to execute the system calls of the operating system.

Table 1. Performance of network I/O when transmitting 100 MB.

Bytes per `read`/`write` operation	Sun's VM [s]		JNuke VM [s]	
	User	Total	User	Total
100	2.3	8.7	35.1	58.1
1,000	0.5	8.7	11.7	19.2
10,000	0.4	8.7	8.9	13.9

We have used the same approach to test application behavior when the target application runs in model checking mode, communicating with external applications. Implementation problems with disambiguating streams across multiple threads have lead to failures for sample applications that involve more than a single message per channel. Work in progress addresses these problems.

5 Related Work

Software model checkers [2,4,6,7,13] store the full program state (or differences to a previously stored state) for backtracking. They are typically implemented as explicit-state model checkers. Milestone creation and rollback operations occur many times during state space exploration. This causes operations to be executed several times when a set of schedules is explored. Such exploration does not treat communication behavior accurately, as described in this paper. One solution is

to model I/O operations as open operations. This abstraction is elegant but generates many spurious behaviors [4,7].

A more general solution to this problem is to lift the power of a model checker from process level to OS level. This way, the effect of an I/O operation is still visible inside the model checker. An existing system that indeed stores and restores full OS states is based on user-mode Linux [10]. That model checker uses the GNU debugger to store states and intercept system calls. The effects of system calls are modeled by hand, but applications can be model checked together without modifying the application code. In that approach, the combined state space of all processes is explored. Our approach analyzes a single process at a time inside a model checker, while running other processes normally. Our approach is therefore more scalable but requires programs to fulfill certain restrictions.

Other virtual machines with such extended replay capabilities have existed before. Initial implementations replayed executions through checkpointing and logging but could not handle system calls [14]. More recent implementations can replay system calls without executing them twice by storing the entire state, including processor registers, before and after such a call [9,12]. They are intended for manual use in conjunction with a debugger, in order to replay one sequence of events. Therefore, they do not allow for systematic state space exploration of a program, which explores a set of states. Furthermore, they use a different approach, intercepting communication at device level, where the network device itself is wrapped. We intercept communication at system call level.

Conventional software model checkers analyze a single process. The results of communication lie outside the scope of such model checkers. Therefore, they cannot be used to model check multi-process applications. One way to solve this problem is to *centralize* a distributed application, i.e., to transform processes into threads [11]. This allows several processes to run in the same model checker, but does not solve the problem of modeling inter-process communication (input/output). Recent work modeled network communication in the centralized model where all processes are executed inside the model checker [1,3].

Centralization model checks multiple communicating processes. However, it may not be possible to execute all processes inside one model checker. To our knowledge, our work is the first approach that allows I/O operations to be carried out by a model checker while still limiting the scope of the model checker to a single process. Furthermore, it even allows model checking of applications where external processes are not running on a platform that the model checkers supports. For example, a server may be written in Java, but clients may be written in a different programming language. While clients cannot run in the model checker VM of JNuke, the server can still be model checked.

6 Conclusions and Future Work

With traditional approaches for model checking software, input/output operations had to be removed by abstraction. Directly resetting program states and executing different branches of a non-deterministic decision is not applicable for

communication, as it interacts with the environment of the program and external applications.

In order to solve this problem, a special rollback semantics for stream-based I/O was defined, which includes network communication. If program behavior is independent of the execution schedule, such a program can be model checked using our caching layer semantics. For protocols that do not require repeated interaction, this semantics was successfully implemented in the JNuke model checking engine.

Future work includes possible relaxations of the completeness criteria defined, regarding the order of I/O operations. Specifically, certain interleaved actions should be allowed, such as log entries.

References

1. C. Artho and P. Garoche. Accurate centralization for applying model checking on networked applications. In *Proc. 21st Int'l Conf. on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006.
2. C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In R. Alur and D. Peled, editors, *Proc. CAV '04*, Boston, USA, 2004. Springer.
3. C. Artho, C. Sommer, and S. Honiden. Model checking networked programs in the presence of transmission failures. In *Proc. 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, Shanghai, China, 2007.
4. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Italy, 2001.
5. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
6. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd Intl. Conf. on Software Engineering (ICSE '00)*, Ireland, 2000. ACM Press.
7. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 174–186, France, 1997.
8. I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proc. 6th Usenix Security Symposium*, San Jose, USA, 1996.
9. S. King, G. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX 2005 Annual Technical Conference*, pages 1–15, Anaheim, USA, 2005.
10. Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato. Model checking of multi-process applications using SBUML and GDB. In *Proc. Workshop on Dependable Software: Tools and Methods*, pages 215–220, Yokohama, Japan, 2005.
11. S. Stoller and Y. Liu. Transformations for model checking distributed Java programs. In *Proc. SPIN 2001*, volume 2057 of *LNCS*, pages 192–199. Springer, 2001.
12. Virtutech. Simics Hindsight, 2005. `http://www.virtutech.se/products/simics-hindsight.html`.
13. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
14. M. Zelkowitz. Reversible execution. *Commun. ACM*, 16(9):566, 1973.