Tools and Techniques for Model Checking Networked Programs

Cyrille Artho
RCIS/AIST
Tokyo, Japan

Watcharin Leungwattanakit
University of Tokyo
Tokyo, Japan

Masami Hagiya
University of Tokyo
Tokyo, Japan

Yoshinori Tanabe
CVS/AIST
Tokyo, Japan

Abstract

For software executing several threads in parallel, testing is unreliable, as it cannot cover all thread schedules. Model checking, however, can cover all possible thread interleavings. Software model checkers can directly verify an implementation, but typically cannot handle network input/output operations, which most programs require. This shortcoming can be addressed by a special model checker designed for multiple processes, or by different kinds of extensions and preprocessors for existing model checkers. This paper surveys currently existing approaches and tools.

1. Introduction

Networked software is complex. It is often implemented as a concurrent program, using threads [29] to handle multiple active communication channels. This introduces two dimensions of non-determinism: Both the thread schedule of the software, and the order in which incoming requests or messages arrive, cannot be controlled by the application. As software testing only covers one particular instance of such a schedule, it is desirable to model check networked software, to ensure that no schedules cause a failure.

Model checking explores the entire behavior of a system under test by investigating each reachable system state [12], accounting for non-determinism in external inputs, such as thread schedules. Recently, model checking has been applied directly to software [3, 6, 10, 13, 15, 16, 30]. 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 again execute certain parts of the program, and thus certain input/output (I/O) operations. However, external processes, which are not under the control of the model checking engine, cannot be kept in synchronization with backtracking.

Figure 1 illustrates the problem. In this example, two threads write a message to their communication channel and then read from it. Both communication channels interact with external processes. Both threads run concurrently, so any interleaving of their operations is possible. Their combined state space is explored inside the model checker, as shown in Figure 2. In that figure, write and read operations are abbreviated by w_i and r_i, respectively, with i denoting the thread ID. As can be clearly seen, execution of all possible interleavings results in multiple backtracking operations, and each communication operation is repeated several times. However, backtracking operations do not apply to peer processes. This results in two problems:

1. The application being model checked will re-send data after backtracking. This will interfere with the correct functionality of external processes.

2. After backtracking, the application will expect external input again. However, an external process does not re-send previously transmitted data.

Thread 1	Thread 2
`void run() { conn1.write("1"); r1 = conn1.read(); }`	`void run() { conn2.write("2"); r2 = conn2.read(); }`

Figure 1. Example program communicating with peer processes.

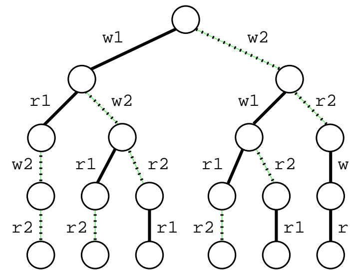

Figure 2. State space exploration inside a software model checker.

Single-process model checkers are therefore incapable of directly handling actions that affect external processes. To address this problem, four approaches are possible:

1. Use stubs to model I/O operations. I/O operations are replaced by stubs that return the same result as the original operation under a given input.

2. Lift the power of a model checker from process level to multiple processes. This way, the entire state of all involved processes is backtracked.

3. Transformation of all involved processes into threads. The transformed, "centralized" program consists of a single process and can be executed in a conventional software model checker.

4. Usage of a special I/O cache that hides backtracking operations in the model checker from external processes. The cache serializes the communication traces in the state space exploration tree.

This paper is organized as follows: Section 2 introduces software model checking. Section 3 surveys existing approaches and tools; Section 4 concludes.

2. Software Model Checking

Model checking [12] explores the entire behavior of a system by investigating each reachable system state. In a classical model checker, both the system and the properties to be checked are translated into finite state machines. Properties are negated, such that the analysis of the state space can detect whether undesired states are reachable. The system starts from an initial state, from which iteration proceeds until an error state is reached, or no further states can be explored. Whenever a non-deterministic choice is reached, all possible successor states are investigated. This iteration can also be performed in the reverse manner, where iteration starts from the set of error states and proceeds backwards, computing whether one of these error states is reachable from an initial state.

Model checking is commonly used to verify algorithms and protocols [18]. However, more recently, model checking has been applied directly to concrete software systems [3, 6, 10, 13, 15, 16, 30]. Software model checking investigates the effects of all possible interleavings between threads and processes involved. The number of interleavings is exponential in the number of operations and threads, resulting in a *state space explosion* for any non-trivial system. For more efficient system exploration, a number of partial-order reduction techniques have been proposed which all have in common that they do not analyze multiple independent interleavings when it can be determined that their effect is equivalent [9, 18]. In software verification, model checking has the advantage that it can automatically and exhaustively verify systems up to a certain size. This contrasts with manual techniques such as theorem proving, which can analyze infinite-state systems but requires human intervention [25].

Properties typically verified in model checking include temporal properties, typically expressed in linear temporal logics [26] or similar formalisms such as state machines [8]. For software, typically checked constructs include pre- and post-conditions such as specified by contracts [21] and assertions. Furthermore, model checkers for modern programming languages typically regard uncaught exceptions and deadlocks as a failure.

3. Approaches and Tools

Different approaches exist that mediate between the model checker and its environment (external processes).

3.1. Stubs

Stubs model I/O operations as open operations that return the same result as the original operation under a given input, or a superset of the original result. This abstraction is elegant but may generate spurious behaviors.

Figure 3 shows how stubs are used in a client-server example. The main process (either client or server) runs in the model checker. Any library functions that interact with the environment are replaced with stubs [6, 10, 14]. The difficulty lies in modeling the environment. In order to produce accurate results, stubs should approximate the behavior of the environment well. At the same time, stubs should be kept simple, lest the complexity of the stubs approaches the complexity of the original program to be abstracted from. However, an abstract stub usually cannot model the outcome of a given operation precisely, and has to allow for a range of possible results. This introduces non-determinism into the system, and may produce spurious outcomes, which are generated by the abstract model but not feasible in the actual (concrete) implementation.

For checking program behavior on a higher level, this approach can be quite successful. High-level properties include robustness of resource management under different outcomes of non-deterministic actions [6, 10, 20]. The exact data transmitted does not have to be modeled for verifying that property. Instead, only a limited range of outcomes, usually success or failure, is assumed. In the extreme case, all control flow constructs dealing with different outcomes are modeled non-deterministically, removing any data involved. In modern programming languages such as Java, control flow includes exceptions. I/O operations either produce a normal outcome, or throw an exception [17]. In conventional testing, the network usually is available, making

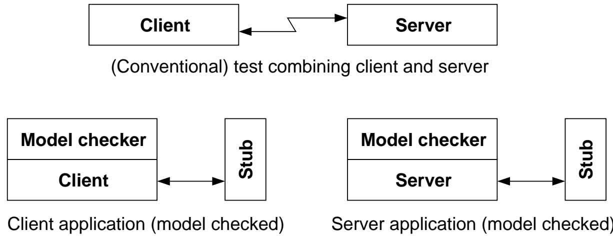

Figure 3. Model checking a client-server system using stubs.

intermittent network failures difficult to test. A stub model focusing on this aspect can find faults in exception handlers that cannot be found through conventional testing [20].

Stubs normally do not interact with the environment. They typically represent a small subset of the functionality of the peer process. While results returned by stubs are usually an approximation, it may be possible to record results of a given test case, as long as operations are deterministic. Recent work has gone into automating this process by a tool that captures communication in a corresponding stub program [7]. For recording, the test run is first executed in a normal environment with all peer processes involved. Information captured in that test run is then replayed by the stub when the target application is run in a model checker.[1]

Stubs can also be approximated iteratively by starting with a very coarse (abstract) version. This version then generates a spurious behavior, which is recognized as such when verifying it on the concrete code. Spurious behaviors are gradually eliminated in a process called *abstraction refinement*. Tools automate such abstraction refinement using a detailed description of the environment and a theorem prover to generate the initial abstract behavior [6, 10, 16].

3.2. Multi-process Model Checkers

Multi-process model checkers allow all involved processes to be the target of model checking (see Figure 4). As the entire state of all involved processes is backtracked, any effects of inter-process communication are backtracked as well. I/O is wrapped by special functions and occurs entirely inside the model checking environment. The generality of the approach makes it very difficult to exploit program properties such as heap symmetry for partial-order reduction. For analysis of Java programs, the Java virtual machine itself would have to be run inside the model checker, which is beyond the capability of current implementations.

[1] In this work, stub creation is performed by an add-on tool that builds on an environment originally designed for centralization [7]. Because of the dual nature of this tool, it is mentioned again in Section 3.3.

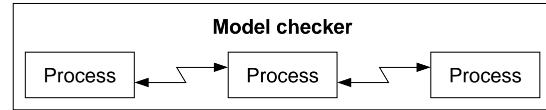

Figure 4. Architecture of a multi-process model checker.

Multi-process model checkers typically consist of several synchronized backtracking execution environments. Processes are executed and backtracked each in their own execution engine, typically implemented using the GNU debugger (GDB). I/O operations are replaced with tool-specific functions, and each processes of a system is run inside a separate debugger that backtracks as needed [11, 16, 23]. These tools have been used to verify system-level communication programs.

Multiple processes can also be controlled by lifting the power of the model checker to operating system level. This way, target processes are allowed to use OS resources or to communicate with other processes. In the preliminary work [24], each target process is run under and controlled by GDB, and the operating system is virtualized using user-mode Linux. GDB processes communicate with an internal monitor running inside user-mode Linux, which communicates with the external monitor running outside user-mode Linux. The external monitor stores and restores full OS states using the snapshot functionality of ScrapBook for User-Mode Linux (SBUML). Consequently, target programs can be verified including their system operations, without any modification.

However, at the time of the preliminary work, many restrictions concerning user-mode Linux and GDB existed. GDB could only control a single thread with the command interface, so only single-threaded target programs could be analyzed. Thus was not possible to run the Java Virtual Machine under user-mode Linux [24].

In addition, in order to make the approach realizable, many kinds of information needed for model checking have to be supplied. First, breakpoints should be appropriately set at target programs where nondeterminism might arise or processes might be blocked. This setting of breakpoints could in principle be automated using static analysis. Second, some additional functions should be added to target programs for inspection by GDB. They include functions to judge whether two internal states of a target process are equivalent, and functions to tell whether a target process is blocked or not. In particular, the former functions should take various properties of a process state, such as heap symmetry, into account, and should be written by hand. Target programs need not be modified, but these functions should be linked with target programs and called by GDB.

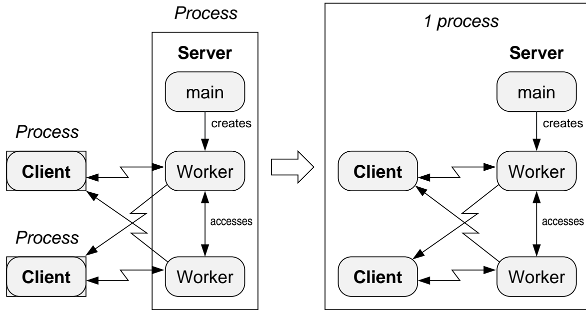

Figure 5. Centralization example: Transformation of three processes into one process.

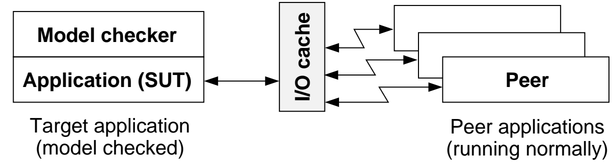

Figure 6. Cache layer architecture.

3.3. Centralization

Centralization executes all processes to be analyzed inside the same model checker. Applications are preprocessed and transformed into a single-process program. The original processes are executed by a wrapper process as threads, inside a single-process model checker (see Figure 5). Certain program transformations are required to maintain separate address spaces for the centralized programs [27]. These transformations can easily be performed manually for typical applications.[2] Centralization covers all possible application behaviors but suffers from the state space explosion problem.

Centralization by itself takes care of executing multiple processes as a single process, but does not handle the underlying network communication. To address this, communication is modeled through a library that provides network-like data streams. Connection management via `accept` and `connect` also has to be modeled correctly. Recent work provided an implementation that correctly simulates TCP/IP connections [1].

Tool support for program centralization of Java programs is available, transforming either Java source code [27], for Java versions up to 1.3, or Java bytecode [1]. The latter tool can be augmented with another transformation simulating I/O communication failures through fault injection [4]. When applying centralization to Java programs, the target software does not require any adaptations. The centralization tool works at a post-compilation stage, transforming a given set of Java applications into a target Java program. The resulting code can directly be executed in any Java runtime environment, such as the standard virtual machine, a debugger, or a Java model checker [1, 4, 27].

Other work has implemented in a similar way, but uses manual instrumentation of communication operations [7]. The tool presented in that work has a mode in which stubs representing peer processes can be run as independent entities (constituting very small centralized processes). If events generated are embedded in the target instead, the result is a stub-based system with the same performance advantage of that approach [7].

As centralization processes the program before it is model checked, any software model checker supporting the given platform can be used. Centralization is also possible for non-Java programs and should be straightforward to implement for similar platforms such as C# [22]. For programming languages that allow pointers to stack data [19, 28], separation of the address spaces of centralized processes may be more difficult. If centralization is implemented entirely as code transformation, then network functionality can also be provided in the target language. The tool approach is therefore the same on other platforms.

[2]Multi-process applications typically have to be abstracted prior to model checking for performance reasons, making the resulting simplified application relatively small.

3.4. I/O Cache

Unlike approaches that execute multiple (possibly transformed) processes inside the model checker, the I/O cache approach only runs a single process in the model checker. A special I/O cache hides backtracking operations in the model checker from external processes [2, 5]. Communication with external processes is physically executed on the host until backtracking occurs. After backtracking, previously observed communication data is fetched from the cache (see Figure 6). This cache is aware of backtracking operations in the model checker [2, 5].

In the I/O cache approach, any communication data sent to or received from the network is kept throughout the entire state space exploration and is not subject to backtracking [5]. However, the communication *position* inside each data stream is local to the process running inside the model checker. It is therefore backtracked whenever that process is backtracked by the model checker [2]. In addition to that, peer processes are polled after each communication event to see whether a response has been sent. This information is used to pair requests and responses correctly, such that the right amount of data is provided after backtracking [2]. This idea is applicable to networked software where communication content does not depend on the global application state. Most programs, such as web servers, can be verified with

this I/O cache, as the response to a request only depends on the request itself, not on previous requests [2].

The I/O cache returns the same communication data that a previously generated stub [7] would return. Unlike stub usage, the caching approach combines recording communication and replaying it in one module, eliminating the need for generating an intermediary stub program. Furthermore, it even allows model checking of applications where external processes are not running on a platform that the model checkers supports [2]. The I/O cache approach is fully automated and can also be adapted to other model checkers and programming languages.

Recent work has implemented the I/O cache approach for the Java PathFinder model checker (JPF) [30]. For that platform, execution of a program inside the model checker does not require any previous transformation or customization. However, a start script that controls all processes involved (outside the model checker) has to be provided. That script has to ensure that client processes do not execute before the server is ready to accept requests. Furthermore, for some applications, the time it takes for a process to respond to a message may have to be customized [2].

3.5. Summary

Table 1 gives an overview of all the tools cited in this paper that are capable of handling inter-process communication. The strengths and weaknesses of each approach can be summarized as follows:

1. Stubs provide an efficient and elegant abstraction of I/O operations. They either have to be written manually, recorded from a previous test run, or generated through abstraction refinement. Abstraction refinement requires a complex tool chain [6, 7, 10, 16].

2. Multi-process model checkers are very powerful. Existing implementations are currently limited by the interface of the GNU debugger [11, 16, 23, 24]. Furthermore, it is extremely difficult to implement library functions or partial-order reduction algorithms such as heap symmetry in such a model checker.

3. Centralization transforms multiple processes into a single process [27]. The approach is elegant and can be fully automated [1]. However, model checking the resulting system suffers from scalability problems, as the number of threads in the resulting system is very large [2].

4. The I/O cache approach is fast, elegant, and applicable to a large family of programs [5]. However, programs that do not fit into the paradigm of a service architecture cannot be verified with this approach [2].

4. Conclusions

Due to the inherent complexity of networked applications, software model checking is tremendously useful for verification. However, input/output operations affect processes outside the model checker. As external processes are not subject to backtracking, the process being model checked cannot communicate directly with its environment. Four solutions to this problem exist: External processes can be replaced by stubs; a special model checker that can support multiple processes can be used; multiple processes can be transformed into a single process; or a special input/output cache can provide a bridge between backtracking communication operations in the model checker and external processes.

References

[1] C. Artho and P. Garoche. Accurate centralization for applying model checking on networked applications. In *ASE 2006*, pages 177–188, Tokyo, Japan, 2006. IEEE Computer Society.

[2] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *Proc. TOOLS 2008*, LNCS, Zurich, Switzerland, 2008. Springer.

[3] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *CAV 2004*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.

[4] C. Artho, C. Sommer, and S. Honiden. Model checking networked programs in the presence of transmission failures. In *TASE 2007*, pages 219–228, Shanghai, China, 2007. IEEE Computer Society.

[5] C. Artho, B. Zweimüller, A. Biere, E. Shibayama, and S. Honiden. Efficient model checking of applications with input/output. *Post-proceedings of Eurocast 2007*, 4739:515–522, 2007.

[6] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *TACAS 2001*, volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.

[7] E. Barlas and T. Bultan. Netstub: a framework for verification of distributed Java applications. In *ASE 2007*, pages 24–33, New York, USA, 2007. ACM.

[8] E. Börger and R. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis*. Springer, 2003.

Tool	Target platform	Technique used to handle networking
CEGAR [10]	C programs	Stubs generated by abstraction refinement; modular checking of each process.
Centralizer [1, 27]	Java programs	Merges multiple Java processes into one, for use with single-process Java model checkers.
CMC [23]	C programs	Control of multiple processes via GNU debugger.
JNuke [3]	Java programs	I/O cache (prototype implementation working for limited examples) [5].
JPF [30]	Java programs	Add-on tools such as centralizer or I/O cache implementation for JPF.
Netstub [7]	Java programs	(1) Centralization; or (2) stubs generated from previously observed program run.
SBUML MC [24]	C programs	Control of multiple processes; host OS runs as process inside user-mode Linux.
SLAM/SDV [6]	Windows device dr.	Abstraction refinement of user-specified environment model.
Verisoft [16]	C programs	Control of multiple processes by restarting. Modular verification also possible [14].

Table 1. Overview of existing tools.

[9] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[10] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[11] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *ICSE 2002*, pages 431–441, New York, USA, 2002. ACM.

[12] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[13] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000*, pages 439–448, Limerick, Ireland, 2000. ACM Press.

[14] J. Dingel. Computer-assisted assume/guarantee reasoning with verisoft. In *ICSE 2003*, pages 138–148, Washington, USA, 2003. IEEE Computer Society.

[15] M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *CAV 2005*, volume 3576 of *LNCS*, pages 148–152, Edinburgh, UK, 2005. Springer.

[16] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL 1997*, pages 174–186, Paris, France, 1997. ACM Press.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

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

[19] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[20] X. Li, H. Hoover, and P. Rudnicki. Towards automatic exception safety verification. In *Proc. FM 2006*, LNCS, pages 396–411, Hamilton, Canada, 2006. Springer.

[21] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[22] Microsoft Corporation. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, Redmond, USA, 2002.

[23] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[24] 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.

[25] D. Peled. *Software Reliability Methods*. Springer, 2001.

[26] A. Pnueli. The temporal logic of programs. In *FOCS 1977*, pages 46–57, Rhode Island, USA, 1977. IEEE, IEEE Computer Society Press.

[27] S. Stoller and Y. Liu. Transformations for model checking distributed Java programs. In *SPIN 2001*, volume 2057 of *LNCS*, pages 192–199. Springer, 2001.

[28] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1997.

[29] A. Tanenbaum. *Modern operating systems*. Prentice-Hall, 1992.

[30] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.