Cache-based Model Checking of Networked Software

Cyrille Artho
RCIS, AIST
Tokyo, Japan

Watcharin Leungwattanakit
and Masami Hagiya
The University of Tokyo
Tokyo, Japan

Yoshinori Tanabe
and Eric Platon
National Institute of Informatics
Tokyo, Japan

Mitsuharu Yamamoto
Chiba University
Chiba, Japan

***Abstract*—Many applications are concurrent and communicate over a network. The non-determinism in the thread and communication schedules makes it desirable to model check such systems. However, a simple state space exploration scheme is not applicable, as backtracking results in repeated communication operations. A cache-based approach solves this problem by hiding redundant communication operations from the environment.**

I. Introduction

Most of the software written today communicates with other software. Networked software is complex. It is often implemented using threads [18] 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 messages arrive, cannot be controlled by the application. In software testing, a given test execution only covers one particular instance of all possible schedules. To ensure that no schedules cause a failure, it is desirable to model check software.

Model checking explores, as far as computational resources allow, the entire behavior of a system under test by investigating each reachable system state [10], accounting for non-determinism in external inputs, such as thread schedules. Recently, model checking has been applied directly to software [5], [6], [8], [11], [13], [14], [19]. 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 operations). However, external processes, which are not under the control of the model checking engine, cannot be kept in synchronization with backtracking. Backtracking would result in repeated communication operations, causing direct communication between the application being model checked and external processes to fail.

We propose a model-checking-aware cache that manages communication between the model checker and its environment [4]. Our approach covers all input/output operations on streams. Our initial work using linear-time cache was applicable to applications that produce a deterministic data stream [3]. Our more recent work introduces new branching-time communication model, which allows for diverging communication traces between different schedules [4]. In cases where the linear-time cache is applicable, our new approach delivers comparable performance. At the same time, we are capable of handling a wider range of protocols and applications.

II. Related Work

Besides caching input/output data, two other major alternative approaches exist [1]:

1) Stubs. Stubs summarize the behavior of the environment, replacing it with a simpler model. The model may be written manually, or recorded from a previous execution to represent the behavior of the environment for a given test case [7]. A stub that over-simplifies the environment may cause false positives or false negatives.
2) Multi-process analysis. The execution environment may be augmented in order to keep the state of multiple processes in sync, for example, by backtracking multiple processes simultaneously [9], [14]. Alternatively, multiple processes may be transformed into a stand-alone system, requiring several program transformations to retain the original semantics [2], [17]. This type of analysis can be very expensive.

III. I/O Caching Algorithm

During the state space exploration, a software model checker backtracks the system under test (SUT). If the SUT communicates with its environment, the problem arises that the SUT is backtracked by the model checker, while the environment is not. This discrepancy between the SUT and its environment can be overcome by caching communication data. A special I/O cache hides backtracking operations, and subsequent repeated communication, from external processes (see Figure 1). Communication with external processes is physically executed on the host until backtracking occurs. After backtracking, previously observed communication data is fetched from the cache [3]. This idea requires an execution environment that is capable of enumerating, storing, and restoring program states; software model checkers that virtualize the execution environment provide this functionality [19].

The I/O cache keeps track of data that has already been sent to or received from the network. It determines if an I/O operation occurs for the first time; if so, data is physically transmitted; otherwise, data is simply read from

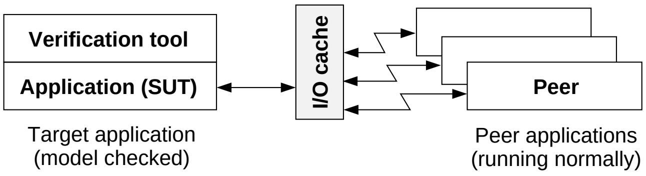
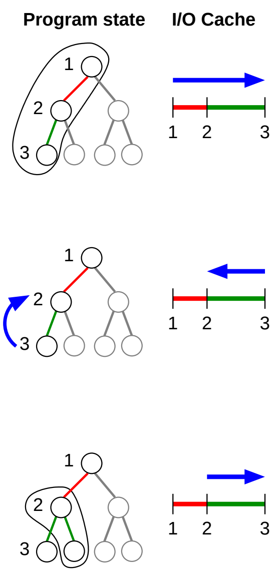


Figure 1. Verification using I/O caching.



New state: I/O data is stored globally. The program state is mapped to the positions of each stream. The size of each message is also stored in a persistent data structure.

Backtracking: The read/write positions in each stream are restored in accordance to the program state. Stream data is kept persistently. This can be regarded as rewinding the position of the stream without erasing it.

Continued exploration: Cached data of previous I/O operations is replayed. Whenever communication data differs from cached data, a new instance of the peer process is created (not shown in this figure).

Figure 2. Mapping program states to communication data.

the cache. Figure 2 illustrates the principle of the caching approach. Communication data is kept persistent by the cache, in conjunction with a mapping of (1) program states to stream positions, and (2) requests to responses [3]. The first mapping allows a reconstruction of the exact stream state upon backtracking; the second mapping determines the size of a response that corresponds to a particular request. After backtracking, the cache replays duplicate responses from memory. It also verifies that duplicate requests are consistent. If a different request is sent, because a different interleaving of threads generates a different output, cached data is no longer valid for the diverging communication trace. To obtain a valid communication trace, a new peer process is launched, and the request is sent to the new instance [4].

IV. Future Work

In future work we will apply cache-based model checking to recent programming models for dealing with concurrency. The generalization of multi-core processors and distributed environments such as "clouds" drive the software industry to build novel abstraction libraries and middleware [12]. These abstraction layers hide low-level detail in concurrency management to simplify development work. These layers usually introduce new semantics, such as a divide-and-conquer approach [12], or domain-specific language constructs [16]. The verification of applications based on such abstraction layers benefits from custom models [15]. We think that such frameworks can also benefit from a cache-based model checking approach.

Acknowledgments: This work was supported by a *kakenhi* grant (2030006) from JSPS.

References

[1] C. Artho. Run-time verification of networked software. In *Proc. RV 2010, Tutorial paper*, volume 6418 of *LNCS*, Malta, Malta, 2010. Springer.

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

[3] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *Proc. TOOLS EUROPE 2008*, volume 19 of *LNBIP*, pages 22–40, Zurich, Switzerland, 2008. Springer.

[4] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto. Cache-based model checking of networked applications: From linear to branching time. In *Proc. ASE 2009*, pages 447–458, Auckland, New Zealand, 2009. IEEE Computer Society.

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

[6] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. 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 *Proc. ASE 2007*, pages 24–33, Atlanta, USA, 2007. ACM.

[8] 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.

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

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

[11] 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. ICSE 2000*, pages 439–448, Limerick, Ireland, 2000. ACM Press.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. OSDI 2004*, pages 137–150, San Francisco, USA, 2004. USENIX Association.

[13] M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *17th Int. Conf. on Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 148–152, Edinburgh, UK, 2005. Springer.

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

[15] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of java-based actor programs. In *Proc. ASE 2009*, pages 468–479, Auckland, New Zealand, 2009. IEEE Computer Society.

[16] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.

[17] 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.

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

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