This is the accepted version of a paper presented at *Workshop on Dependability of Network Software Applications (DNSA 2010)*.

Citation for the original published paper:

Artho, C., Leungwattanakit, W., Hagiya, M., Platon, E., Tanabe, Y. et al. (2010)
Cache-based Model Checking of Networked Software.
In: *Proc. Workshop on Dependability of Network Software Applications (DNSA 2010)*

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-199137
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...
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


