Run-time Verification of Networked Software

Cyrille Artho
c.artho@aist.go.jp

Research Center for Information Security (RCIS),
National Institute of Advanced Industrial Science and Technology (AIST),
Tokyo, Japan

Abstract. Most applications that are in use today inter-operate with other applications, so-called peers, over a network. The analysis of such distributed applications requires that the effect of the communication with peers is included. This can be achieved by writing or generating stubs of peers, or by including all processes in the execution environment. The latter approach also requires special treatment of network communication primitives.
We also present an alternative approach, which analyzes a networked application by recording and caching its communication with peers. Caching becomes useful when several traces of the application are analyzed. It dispenses with the need of generating a new peer application execution for each different execution of the main application. Such a caching framework for input/output has been implemented on the Java PathFinder platform, which can be used to verify executions of non-deterministic applications at run-time.

1 Introduction

Most of the software written today does not implement a stand-alone system, but communicates with other software. Testing such networked software requires that either all necessary applications be running, or that applications outside the scope of the analysis be replaced by an open environment, or stubs. An open environment non-deterministically returns all possible outcomes of a given function, and is often used for analysis on a more abstract level. In model checking, approaches exist that iteratively narrow down the scope of an open environment, to generate a result that mirrors actual executions more closely [7,12,13]. Nonetheless, such techniques may not always generate over-approximations that are precise enough to analyze complex systems.

Run-time verification takes a different approach to analysis, executing the actual system under test (SUT) rather than an approximation. This has the benefit that observed execution failures always correspond to actual system failures. However, the analysis of a few execution traces may miss certain defects if a system is not fully deterministic. In particular, concurrent systems suffer from this problem: The thread schedule is not controlled by the application, and may vary between executions. Classical testing invariably covers only a subset of all possible schedules and may miss defects.

For concurrent software, run-time verification provides several means of extending classical testing. Approaches exist that analyze the detailed behavior of software, for example, its lock usage, in addition to its output [27]. Other approaches observe several execution traces that are generated in a way as to maximize the potential of finding faults [9,18]. For smaller systems, the coverage of possible execution schedules may even be exhaustive [22,30]. Exhaustive techniques are at the boundary between model checking and run-time verification, analyzing concrete executions while striving to cover all possible outcomes.

If one attempts to apply such techniques to networked software, the challenge of orchestrating the execution of a distributed system arises. Multiple executions may be achieved by restarting the SUT, or by backtracking it to a previous state. In either case, after the SUT has been backtracked or restarted, its state may not be logically consistent anymore with the state of its peer processes. Distributed systems can therefore not be directly executed when using backtracking.

1.1 Overview

This tutorial presents several approaches for verifying networked software [4]:

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 [8].
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 [13,19]. Alternatively, multiple processes may be transformed into a stand-alone system, requiring several program transformations to retain the original semantics [2,29].
3. Caching. Communication between the SUT and its environment is observed, and a model of observed communication traces is generated. This model can then be used to replay communication on demand for subsequent repeated executions of the SUT [3]. Caching yields major performance benefits if different outcomes of the SUT are analyzed by backtracking, thus replaying subsets of the full execution many times [5]. Challenges in this approach include tracking message boundaries while having only an incomplete knowledge of peer processes [3], and handling non-deterministic input/output of the SUT [5].

1.2 Outline

This text is organized as follows: Section 2 shows how software model checking relates to run-time verification. Problems arising with distributed systems are covered in Section 3. The three approaches presented in this tutorial are covered in Sections 4, 5, and 6. Section 7 concludes.

2 Run-time Verification and Software Model Checking

Model checking [14] explores the entire behavior of a system by investigating each reachable system state. In classical model checker, both the system and the properties to be checked are translated into finite state machines. The properties are negated in the process, such that the analysis of the state space can detect whether undesired states are reachable. The system starts in an initial state, from where iteration proceeds until an error state is reached, or no further states are to be explored. 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 [23]. However, more recently, model checking has been applied directly to concrete software systems [6,7,12,15,17,19,30]. Software model checking investigates the effects of all non-deterministic choices in the SUT, and in particular, 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 a more efficient system exploration, a number of partial-order reduction techniques have been proposed. They have in common that they do not analyze multiple independent interleavings when it can be determined that their effect is equivalent [11,23].

Properties typically verified in model checking include temporal properties, typically expressed in linear temporal logics [26] or similar formalisms such as state machines [10]. For software, typically checked constructs include pre- and post-conditions such as specified by contracts [25] and assertions. Furthermore, software model checkers for modern programming languages typically regard uncaught exceptions and deadlocks as a failure.

In software verification, model checking has the advantage that it can automatically and exhaustively verify systems up to a certain size. If the state space of the SUT becomes too large, a possible solution is to prioritize the search of the state space towards states that may more likely uncover defects. User-defined heuristics guide the state space search towards such states. This type of analysis may be implemented in a software model checker framework [21] or in the standard run-time environment, by choosing a heuristic that likely uncovers new thread schedule with each program execution [9,28].

In this sense, the two domains have much in common. Both software model checking and other run-time verification tools analyze the actual implementation of the SUT (or a derived version of it that preserves the original run-time behavior). Both techniques cover a set of execution traces that characterizes a large part of the behavior of the SUT, but not necessarily the entire state space.

In this paper, the term *backtracking* will denote the restoration of a previous state, even if that state is not a predecessor state of the current state. This definition allows the term "backtracking" to be used for search strategies other than depth-first search, and for techniques where a previous system state is restored by re-executing the SUT again from its initial state with the same input (and thread schedule) up to the given target state.

3 Distributed Applications

The analysis of multiple execution traces of a SUT becomes challenging if the SUT communicates with external processes. Backtracking the SUT allows the restoration of a previous state without having to execute the system again up to a given state. However, when backtracking the SUT, external (peer) processes are not affected. As a consequence of this, the states of the SUT is likely no longer consistent with the states of peer processes. From this inconsistency, two problems arise [3]:

1. The SUT will re-send data after backtracking, which interferes with peers.
2. After backtracking, the SUT will expect the same external input again. However, a peer does not re-send previously transmitted data.

Some run-time environments may be able to control the states of multiple processes at the same time. For example, a hypervisor can execute the entire operating system inside a virtual machine, and store and restore any state. However, such tools are, at the time of writing, slow for the usage of state space exploration because of the large size of each system state (consisting of an entire operating system at run-time). Furthermore, processes running on external systems cannot be handled on this way. We therefore focus on dealing with distributed (networked) software executing on verification tools that support one process at a time. In this context, the SUT will denote the process to be verified, and a *peer process* denotes another application running outside the scope of the verification tool. The *environment* of the SUT consists of several peer processes, and other resources used by the SUT, such as communication links to peers.

4 Modeling External Processes as Stubs

If an external process cannot be controlled by an analysis tool, a possible approach is to exclude it from analysis, and replace it with an open model that represents all its possible behaviors. Such an abstract environment model has been successfully used in software model checking [7,12,16]. When targeting complex applications, though, an open model may include too many behaviors of the environment to make analysis tractable. Furthermore, for run-time verification, concrete executions of environment processes are needed, as the SUT cannot be executed against an open model at run-time.

In the case of networked programs, any interaction between the SUT and its environment occurs through the application programming interface (API) providing network access. Responses of an environment process are also retrieved through this API. This allows a replacement of the API with a light-weight skeleton, or *stub,* which only returns the appropriate response of the environment process, without actually communicating with it. The open model is therefore closed with a specialized implementation that is tailored to one particular verification run (or a limited number of tests). Compared to the actual environment process, the implementation of the stub can usually be simplified significantly,

removing operating system calls and inter-process communication. The implementation of such a stub is often simple enough to be written manually. For larger projects, frameworks exist that implement some of the common aspects of network APIs [8]. Another approach is to record a communication trace between the SUT and its environment for a given test run, and then generate a stub that implements the recorded responses [8].

In some cases, not all responses from a peer process may be fully deterministic. Network communication involves inherent non-determinism: Even during the verification of a concrete execution with deterministic input, it is possible that network communication is severed due to transmission problems. The reason for this lies in possible transmission problems and is not visible in the SUT. From the point of view of software, communication failures may be regarded as a non-deterministic decision of the environment. As a result, an exception indicating the loss of connectivity may be thrown at run-time, as an alternative outcome to the expected response.

Such exceptions cannot always be tested easily using conventional unit or system tests. A stub is quite suitable for modeling such outcomes, though. In the stub model, one execution for the successful case and another execution for the failure case can be covered. For verification, one can either use a software model checker that interprets such non-determinism as two related test executions, backtracking when necessary [30], or use a run-time verification tool that analyzes execution traces and then selectively implements fault injection, covering both outcomes [1].

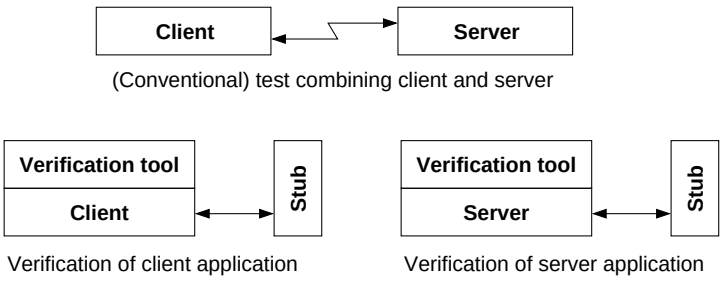


Fig. 1. Verification using stubs.

Finally, an approach using stubs for peers is unlikely to find defects in peer processes. When using stubs to analyze a distributed system, one system is analyzed at a time (see Figure 1). It is therefore advisable to alternate the roles of SUT and stubs. Even then, a stub that replaces the other processes during verification may not reflect all possible outcomes of a peer process, especially when a stub is synthesized from recording one sample execution. This limits the degree of confidence gained. Nonetheless, stub-based verification is an elegant and efficient way of analyzing a distributed system, and works especially well

when the target is concurrency within the SUT, and fault injection for the interaction between the SUT and the environment. The simplification of peers usually removes interleavings between the SUT and peers, which can be regarded as a partial-order reduction. The performance gained by this abstraction enables the usage of techniques such as fault injection or software model checking, in cases when they may not scale up to multi-process systems in their entirety.

5 Centralization

Many existing software analysis tools can only explore a single process and are not applicable to networked applications, where several processes interact. More often than not, extending the capabilities of these systems towards multiple processes would take considerable effort. It is often easier to reverse the problem, and transform multiple processes into a single process that behaves the same as the original (distributed) application. This approach is called *centralization* [29].

For centralization, processes are converted into *threads* and merged into a single process. Networked applications can then run as one multi-threaded process. Figure 2 illustrates the idea: All applications are run inside the same process, as threads. I/O between applications has to be virtualized, i. e., modeled such that it can be performed inside the execution environment. In the remainder of this section, the term "centralized process" will denote all threads of a given process that was part of a distributed system. In that terminology, three processes are centralized in the example in Figure 2, and converted into one physical process.

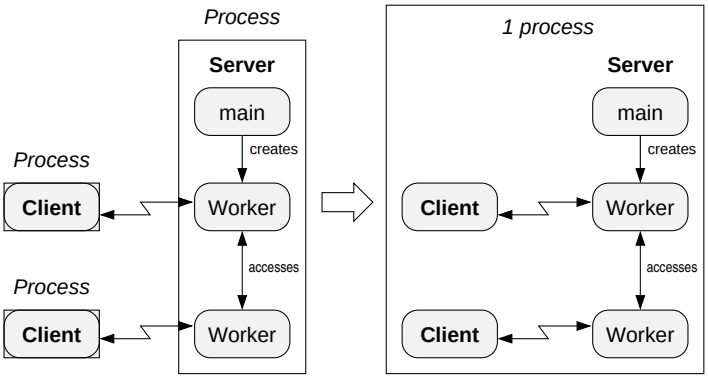


Fig. 2. Centralization.

In the remainder of this section, we discuss the treatment of Java [20] programs. The ideas presented here can be readily generalized to other platforms. Centralization of a Java program involves four issues [2,29]:

1. Wrapping applications (processes) as threads, and starting them as such.

2. Keeping the address space of each process separate. In object-oriented languages, this is not a problem for normally created instances, as they are created separately for each application. Therefore, this problem is reduced to the management of global variables, which are contained in the data segment of C or C++ programs, and in static fields in Java. In Java, each static field is unique and globally accessible by its class name. This uniqueness applies per class and thus per VM. In the centralized version, field accesses to static fields are shared between centralized processes, must be disambiguated. In addition to that, access to environment variables may have to be wrapped to present a distinct environment to each centralized process.
3. Static synchronized methods. In Java, instance-level synchronization is performed implicitly by the VM whenever a method is `synchronized`. For static methods, synchronization accesses a class descriptor that should again be unique for each centralized process. In programming languages that have no built-in concurrency constructs, like C or C++, global locks are already transformed in the previous step.
4. Shutdown semantics. When a process is shut down, its run-time environment closes open sockets and files before exiting. Furthermore, background (daemon) threads are killed. These actions do not occur if just a thread terminates. After centralization, such clean-up actions therefore do not occur automatically anymore, unless all centralized processes have terminated. Likewise, actions that terminate an entire process would end up terminating all centralized processes in the centralized version; this has to be prevented to retain the original semantics.

Figure 2 illustrates the overall approach on a typical client-server example. Clients are single-threaded and communicate with the server. The server uses one main thread to accept requests, and one worker thread per request to handle accepted requests. Worker threads on the server side share a global state, such as the number of active connections. Centralization transforms the given processes into one process. In most cases, this transformation results in an additional wrapper thread that launches the main thread of each process involved.

Once all applications have been centralized, the effects of network communication have to be internalized, such that they can be analyzed entirely within the memory space of the centralized program. In this transformation, blocking and unblocking calls, and bidirectional communication, have to be modeled such that their semantics are preserved. The remainder of this section covers the necessary program transformations to address the four points listed above, and the treatment of network communication in the resulting transformed program.

5.1 Program transformations for centralization

The four points above address two underlying problems: combining all processes into a single process, and adapting the resulting single-process system such that it exhibits the same behaviors as the original multi-process system.

The first challenge of the required program transformation is to wrap the `main` method of each centralized process in its own thread (see Figure 3). The

wrapper code constructs an instance of `CentralizedProcess`, which extends the built-in thread class with a virtual process ID. This process ID is used later on to distinguish the address spaces of the centralized processes [29]. Each application is called with its original arguments and receives a distinct process ID. In Figure 3, the exact arguments to `main`, and the code that ensures that the server is ready to accept incoming client requests, are elided.

```
 1 /* Wrapper for running a process inside a thread */
   public class CentralizedProcess extends Thread {
     public int pid;

 5   public CentralizedProcess (int procId) {
       super();
       pid = procId;
   } }

10 /* Wrapper for combining all processes */
   public class LaunchProcesses {
     public static final void main(...) {
       new CentralizedProcess(0) {
         public void run() {
15         Server.main(server_args);
       }}.start();

       // wait for server to be ready

20     for (int i = 1; i <= N; i++) {
         new CentralizedProcess(i) {
           public void run() {
             Client.main(client_args);
         }}.start();
25 } } }
```

Fig. 3. Wrapping and launching centralized processes.

Second, in the implementation of the SUT, access to global data has to be changed. Code belonging to distinct applications must not (inadvertently) access the same memory location when centralized. Such a disambiguation of data accesses can be achieved by changing each global variable to an array, using the virtual process ID as an index to that array [29]. This transformation can be automated by tools that rewrite source code or byte code [2,29]. For complex data structures in Java, care has to be taken that code to initialize the resulting arrays is generated. For example, an integer field is set to 0 by default in Java, but an array is not created without corresponding code to create it. The initialization of array entries to 0 is again automatic in Java.

Third, it is possible in Java to use class descriptors for locking. Class descriptors can only exist once in each run-time environment, so the approach described above to replicate normal data structures is not applicable in this case. The solution is to use *proxy locks* instead of a class descriptor [29]. One array of proxy

locks is created for each class descriptor used for locking. Proxy locks are accessed by virtual process ID as described above. Care has to be taken that class descriptors are not replaced when they are used for the purpose of reflection. In that case, the actual class descriptor, which is unique even in the centralization version of the program, has to be used. The distinction of the two cases, followed by code transformation, can usually be made by data flow analysis [2,29].

Finally, the semantics of program shutdown should be reflected accurately. There are two sides of this problem: On the one hand, a call to `exit` terminates only one process in the original application, but all centralized processes in the centralized program. On the other hand, resources such as files or network connections should be closed in the centralized version even if the run-time environment has not terminated yet.

In Java, the first aspect of shutdown semantics can be addressed by changing calls to `System.exit` to throwing an instance of `ThreadDeath`, which terminates the active thread. Complex cases may need code that manages the number of active child threads per centralized process, as this necessary to determine when a process has terminated. This is not always trivial in a centralized program, and an automatic transformation may not always be possible; for example, in Java, there is no direct way to kill one thread from another thread [20].

The second aspect, the automated release of shared resources, can be implemented by writing a custom shutdown handler, which is invoked whenever the last thread belonging to a centralized process terminates. Both aspects of the shutdown semantics require extensive run-time data structures to keep track of the status of each process, and are work in progress [2].

5.2 Networking for the centralized program

Distributed applications need communication mechanisms to interact. Such communication includes the usage of files or shared buffers. These can be modeled using a shared global array in the centralized version. More typically, though, communication takes place over a network. Inter-process communication mechanisms involve low-level operating system calls and are often outside the scope of run-time verification tools. While centralization itself makes multiple processes visible to a single-process analysis tool, it is also necessary to make inter-process communication transparent. This can be achieved by providing a communication model library. The library takes advantage of centralization, and provides the original communication API while sending messages between threads rather than (possibly remote) processes. Using this model library instead of the default library, inter-process communication takes place entirely within the memory of one application.

While this section only describes network communication in detail, the principles described are also applicable to other types of inter-process communication. The common aspects are as follows:

1. In an initial phase, applications set up a communication link. This usually involves one process waiting (listening) for another process to connect. Within

each process, both actions are blocking, and will suspend the current thread performing this action until the action has completed.
In the centralized program, blocking system calls that require a response from another process are modeled with inter-thread signals. In Java, `wait/notify` pairs in both threads involved, model the "handshake protocol" between centralized processes.

2. Once communication is established, a bidirectional channel is available for the transmission of messages. Data that is communicated between applications can be modeled with constructs that share data between threads, such as arrays or inter-thread pipes.

For simplicity, network communication is described here as an interaction between two centralized processes, a *client* and a *server*. The server accepts incoming connections at a certain port. The client subsequently connects to that port. After a connection is established, a bidirectional communication channel exists between the client and the server. Communication can then be performed in an asynchronous manner: Underlying transport mechanisms (commonly TCP/IP) ensure that sent messages arrive eventually (if a connection is available), but with some delay. This applies to messages in both directions. A connection can be closed by the client or the server, terminating communication.

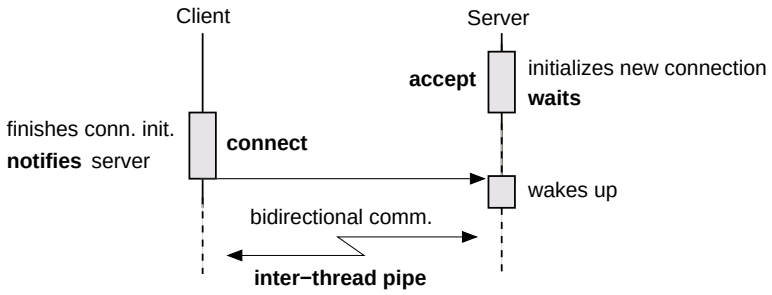


Fig. 4. Client-server communication.

For establishing the network connection, we use a two-step initialization (see Figure 4). In the first step, the `accept` call of the server, the server sets up its part of the connection and then blocks (waits) on a common semaphore, which exists in the network model code. When the client calls `connect`, it completes its part of setting up the connection, and then unblocks (notifies) the server. This ensures that the sequence of each original application passing through blocking library calls is preserved in the centralized version. Upon connection, two unidirectional inter-thread pipes are set up, as available through `java.io.PipedInputStream` and `java.io.PipedOutputStream`. They model the underlying network communication normally provided by system libraries, replacing inter-process communication by inter-thread communication [2].

Once the network model for the centralized application is available, the code that starts the centralized clients after the server is ready, can be provided (see Figure 3). By inspecting the state of the connection hand-shake, the wrapper code sees if the server has partially initialized its first connection, and is able to accept an incoming client request. At that point, the execution of the wrapper code can continue, and the clients can be launched [2].

To summarize, centralization of networked software consists of program transformation, and a network model library. The resulting centralized application can be executed by any run-time environment, making the approach very versatile for verification. While the complexity of all processes combined may be exceed the capabilities of a heavy-weight analysis tool, centralization is a promising technique for light-weight run-time verification algorithms, extending the scope of single-process tools (such as debuggers) to multiple processes.

6 Input/Output Caching

Unlike approaches that execute multiple (possibly transformed) processes inside the analysis tool, it is possible to execute only one process in the analyzer, and mitigate the effects of backtracking by caching the input/output (I/O) of the SUT. This *I/O cache* approach only runs a single process using the verification tool. Other processes run in their normal environment, perhaps even on remote hosts that are not controlled by the test setup. If multiple communication traces of the SUT are generated by backtracking the state of the SUT, followed by a different scheduling choice, then the state of peer processes has to be kept consistent with the SUT. Without enforcing consistency after backtracking, the state of the SUT would no longer correspond to the state of the communication protocol, as communication has taken place in the physical world and cannot be backtracked.

This discrepancy between the state of the SUT and the physical world can be overcome by caching communication data. A special I/O cache hides backtracking operations, and subsequent repeated communication, from external processes (see Figure 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 [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 [30].

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 the cache. Figure 6 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

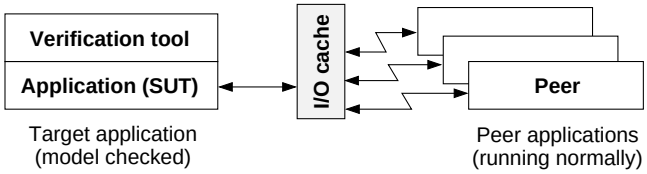
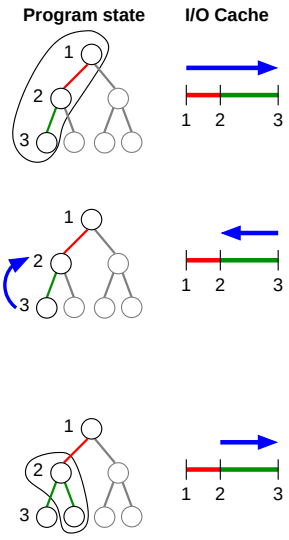


Fig. 5. Verification using I/O caching.

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 [3].



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 current read/write positions in each stream are restored in accordance to the program state, but 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. Output data is compared to previously cached data. Whenever communication data differs from cached data, or exceeds it, a new instance of the peer process is created. Previously cached data is replayed up to the given point, after which a new branch in the data cache is created (not shown in this figure).

Fig. 6. Mapping program states to communication data.

Communication diverges in cases where requests depend on a global program state, for example, when the value of a global counter is sent over the network. When communication diverges after backtracking, the state of peer processes is no longer consistent with the state of the SUT. In such cases, peer processes have to be reset to a state that is equivalent to the state of the backtracked SUT. An equivalent peer state can be obtained by sending the new (diverging) input to a peer process, starting from the state at which communication diverged. This requires a new copy of the peer program, running from the point where communication diverges.

An extended cache model starts a new peer process in such cases, and replays communication data up to the point before the trace diverged. This results in a tree structure of communication traces. Despite the need to restart peer processes occasionally, the cache-based solution is still far more efficient than approaches where the peer processes are restarted each time after backtracking [5]. Work is in progress to replace restarting peer processes with restoring a recorded snapshot of all peers, by using virtualization tools [24].

The I/O caching approach analyzes one application in a distributed framework at a time. The other applications run normally. When analyzing a client, the service it requests may even be hosted on a remote machine that is controlled by the verification setup. When analyzing a server, the verification environment has to be able to execute a client on demand, to allow the server to receive requests. In either case, peer processes are not aware that the SUT is not executing serially, but subject to backtracking. Peer processes can therefore be executed in their normal test environment. The cache enables the verification tool to use backtracking to verify the outcome of non-deterministic decisions of the SUT, without always having to restart all peer processes involved after backtracking. As only the SUT is subject to backtracking, the caching technique ignores non-determinism in peer processes. Therefore, the technique is potentially unsound, but this unsoundness comes at a vast improvement in scalability compared to sound approaches such as centralization [3].

7 Conclusion

Distributed applications consist of several processes interacting over a network. Many existing analysis tools are designed to explore the state space of only a single process. Luckily, there exist several ways to adapt a multi-process program to a single-process analysis tool.

One approach is to treat each application separately. Interactions with other applications can be simulated by *stubs,* which replace the original function call and return a value suitable for testing. Stubs can be written manually or synthesized from data recorded in a sample execution. The resulting system is simpler than the original program. For concurrent peer processes, stubs generated from sample executions provide unsound but efficient verification.

To fully verify a distributed system in a single-process environment, multiple processes can be *centralized,* converted into a single process. In such a conversion, distinctive features of separate processes, in particular, their separate address spaces, have to be preserved. Finally, a new implementation of the network API is needed for the centralized program, where inter-process communication is replaced by inter-thread communication. The resulting program is fully self-contained, and all effects of communication are visible inside a single process.

As another alternative, network communication can be captured and replayed on the fly. This requires a *caching system* that provides transparent interaction between the system under test and external processes. The cache has to be inte-

grated in the analysis tool, though, and the approach may be unsound. However, it provides the performance advantage of stubs without requiring code synthesis.

Input/output caching requires a special run-time environment, but it has the advantage of providing a virtual network environment that can communicate with external peers. The other approaches can be used without requiring adaptations of the verification tools, making it possible to verify multi-process systems on tools that handle only one process by themselves.

References

1. C. Artho, A. Biere, and S. Honiden. Exhaustive testing of exception handlers with enforcer. *Post-proceedings of 5th Int. Symposium on Formal Methods for Components and Objects (FMCO 2006)*, 4709:26–46, 2006.
2. C. Artho and P. Garoche. Accurate centralization for applying model checking on networked applications. In *Proc. 21st Int. Conf. on Automated Software Engineering (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, and Y. Tanabe. Tools and techniques for model checking networked programs. In *Proc. SNPD 2008*, Phuket, Thailand, 2008. IEEE.
5. 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. 24th Int. Conf. on Automated Software Engineering (ASE 2009)*, pages 447–458, Auckland, New Zealand, 2009. IEEE Computer Society.
6. C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. 16th Int. Conf. on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.
7. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.
8. E. Barlas and T. Bultan. Netstub: a framework for verification of distributed Java applications. In *Proc. 22nd Int. Conf. on Automated Software Engineering (ASE 2007)*, pages 24–33, Atlanta, USA, 2007. ACM.
9. Y. Ben-Asher, Y. Eytani, and E. Farchi. Heuristics for finding concurrent bugs. In *Proc. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003)*, page 288a, Nice, France, 2003.
10. E. Börger and R. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis*. Springer, 2003.
11. D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.
12. 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.

13. S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proc. 24th Int. Conf. on Software Engineering (ICSE 2002)*, pages 431–441, New York, USA, 2002. ACM.
14. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
15. 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 Int. Conf. on Software Engineering (ICSE 2000)*, pages 439–448, Limerick, Ireland, 2000. ACM Press.
16. J. Dingel. Computer-assisted assume/guarantee reasoning with VeriSoft. In *Proc. 25th Int. Conf. on Software Engineering (ICSE 2003)*, pages 138–148, Washington, USA, 2003. IEEE Computer Society.
17. 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.
18. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. 20th IEEE Int. Parallel & Distributed Processing Symposium (IPDPS 2003)*, page 286, Nice, France, 2003. IEEE Computer Society Press.
19. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL 1997)*, pages 174–186, Paris, France, 1997. ACM Press.
20. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.
21. A. Groce and W. Visser. Heuristics for model checking Java programs. *Int. Journal on Software Tools for Technology Transfer*, 6(4):260–276, 2004.
22. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.
23. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
24. W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto. Introduction of virtualization technology to multi-process model checking. In *Proc. 1st NASA Formal Methods Symposium*, pages 106–110, Moffett Field, USA, 2009.
25. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.
26. A. Pnueli. The temporal logic of programs. In *Proc. 17th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57, Rhode Island, USA, 1977. IEEE, IEEE Computer Society Press.
27. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
28. S. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. 2nd Int. Workshop on Run-time Verification (RV 2002)*, volume 70(4) of *ENTCS*, pages 143–158, Copenhagen, Denmark, 2002. Elsevier.
29. S. Stoller and Y. Liu. Transformations for model checking distributed Java programs. In *Proc. 8th Int. SPIN Workshop (SPIN 2001)*, volume 2057 of *LNCS*, pages 192–199. Springer, 2001.
30. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.