Cache-based Model Checking of Networked Applications: From Linear to Branching Time

Cyrille Artho*, Watcharin Leungwattanakit†, Masami Hagiya†, Yoshinori Tanabe†, Mitsuharu Yamamoto‡
*Research Center for Information Security, AIST, Tokyo, Japan
E-mail: c.artho@aist.go.jp
†Department of Computer Science, University of Tokyo, Tokyo, Japan
E-mails: {watcharin,hagiya}@is.s.u-tokyo.ac.jp,y-tanabe@ci.i.u-tokyo.ac.jp
‡Department of Mathematics and Informatics, Chiba University, Chiba, Japan
E-mail: mituharu@math.s.chiba-u.ac.jp

***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. In this work, we propose a change from a linear-time to a branching-time cache, allowing us to relax restrictions in previous work regarding communication traces that differ between schedules. We successfully applied the new algorithm to real-life programs where a previous solution is not applicable.**

***Keywords*-Software model checking; software verification; networking; input/output; caching**

I. Introduction

Networked software is complex. It is often implemented using threads [26] 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. 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 [12], accounting for non-determinism in external inputs, such as thread schedules. Recently, model checking has been applied directly to software [5], [7], [10], [13], [15], [16], [28]. 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 [2]. Our approach covers all input/output operations on streams. Previous work using linear-time cache was applicable to applications that produce a deterministic data stream [2]. We introduce a new branching-time communication model, which allows for diverging communication traces between different schedules. 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.

This paper is organized as follows: Section II introduces our algorithm, while Section III formalizes it. Experiments are given in Section IV. Section V describes related work. Section VI concludes this paper and outlines future work.

II. Intuition of the Caching Algorithm

When analyzing a multi-threaded program, a model checker explores all non-deterministic decisions in that program. Alternative schedules are explored by backtracking to a previously stored program state, running the program again from that state under a different schedule.

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. Let "system under test" (SUT) denote the application executing inside the model checker. Execution of the SUT is subject to backtracking. External processes are called *peers* and can implement either client or server functionality, as defined in [27]. A *request* is a message written (sent) to a peer, and a *response* a message read (received) from it. I/O denotes such input/output.

A. Handling redundant actions after backtracking

Effects of I/O operations cannot be reversed by backtracking, as the environment of the system is affected.

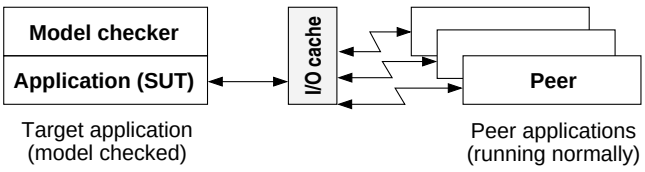

Figure 1. Cache layer architecture.

When model checking an application that communicates with peers, peers are not backtracked. Two problems arise:

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

Our approach executes a single process inside the model checker, and runs all peers externally. It uses a cache to relay data between the model checker and its environment (see Figure 1). Redundant externally visible operations, such as input/output, have to be hidden from external processes. The cache layer intercepts any network traffic and represents the state of communication between the SUT and peers 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. Whenever communication proceeds beyond previously cached information, new data is both physically transmitted over the network and also added to the cache.

Peer processes may run on external hosts and require features such as database access that a given model checker cannot support. We assume that testing is automated and starts from a well-defined initial state each time. However, even when automated test frameworks are used, it may be infeasible or inefficient to backtrack peer processes. Our approach considers each peer as a black-box process and exhaustively searches the state space of only *one* process at a time. To verify all processes, model checking has to be applied once to each type of process, with other processes running as peers.

B. Non-determinism

In our analysis of non-determinism, we focus on observable communication behavior. As far as the input and output can be observed, we consider a system to be deterministic iff a unique input sequence produces a unique output sequence. For any two runs with equal inputs i, i', the observable outputs o, o' match:

$$\forall k, (\forall j < k \cdot i_j = i'_j) \rightarrow o_k = o'_k. \quad (1)$$

In this definition, requests and responses have length 1. Non-unit message lengths can be dealt with by observing peer behavior, as specified in earlier work [2]. Definition 1 applies to a single communication channel. It assumes that all data sent and received over that channel is totally ordered. For a SUT communicating with peers, we consider a peer to be deterministic if it sends a deterministic response on each connection, according to Definition 1 (inputs correspond to requests of the SUT to a peer, outputs correspond to responses sent by the peer to the SUT).

In the SUT, I/O operations of multiple connections may be interleaved, and thus the occurrence of all I/O operations over all channels is only partially ordered. If a system is deterministic according to Definition 1, then there exists a total order of I/O operations on each channel even if no total order exists over all operations on all channels.

C. From linear time to branching time

Previous work used a linear-time cache structure to store communication data, and matched each response to its preceding request by polling the corresponding peer process [2]. The assumption was that operations on a given communication channel are totally ordered and repeatable. Specifically, two properties have to hold for a linear-time cache to be applicable:

1) Deterministic peer responses, as per Definition 1: For each communication channel, for a given sequence of requests, the corresponding sequence of responses is deterministic. This entails that if a given trace is replayed against a peer, the peer will exhibit the same behavior that was previously observed. Peers may be multi-threaded and communicate among each other as long as they are deterministic.
2) Consistent application behavior: For each thread and each socket, the same requests are issued regardless of the thread schedule. This assumes that as far as the output sequence of the SUT can be observed, the two output sequences o, o' of a SUT on a given communication channel always match:

$$\forall j \cdot o_j = o'_j. \quad (2)$$

The first requirement (Definition 1) is less restrictive. If only one possible response for each request sequence (input to the peer) is analyzed, all runs analyzed still correspond to possible real executions, although alternative peer responses (outputs) may be missed. In some cases, our approach remains applicable but becomes unsound, as certain failures may only be provoked by particular peer responses that are not observed. In other cases, a given peer execution may not be repeatable due to non-determinism, even for the same request/input to the peer; for such peers, a cache-based approach is not applicable [4].

The second restriction (Definition 2) is more limiting. If the behavior of the SUT after backtracking deviates from previously observed behavior, then the linear cache model is no longer adequate. Model checking has to be aborted in

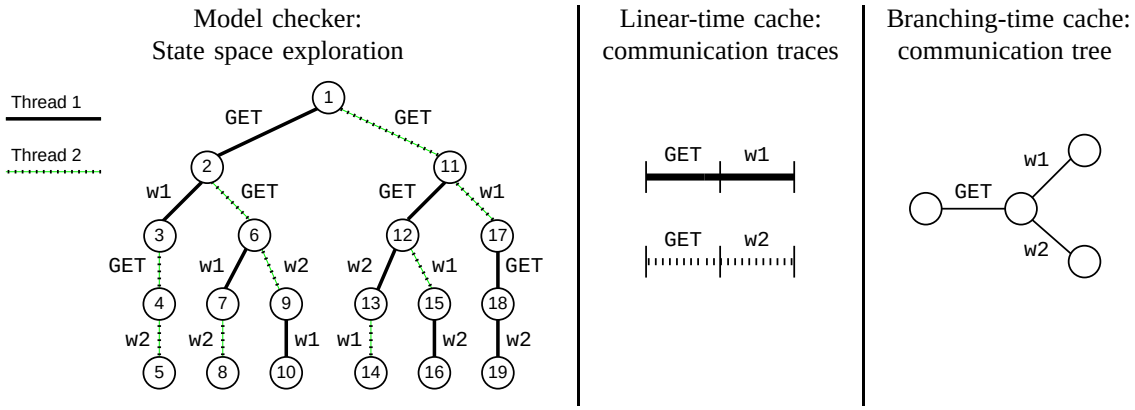

Figure 2. State space in the model checker (left) and cached communication data (middle/right).

such cases [2]. In our experiments, we have found that the restriction is usually hit within a few seconds of state space exploration. For many types of programs, the old approach can therefore only cover a small percentage of the entire state space.

This work replaces the linear-time cache with a branching-time cache. By doing so, we can drop the second restriction. Relaxation of the first restriction would correspond to permitting non-determinism in peer processes, or the existence of a global state involving the entire system state of all communicating processes [2]. Such systems can be model checked by controlling all processes involved in one model checker, for example, by merging several processes into a single process [1], [25]. Usage of our technique in these cases corresponds to an abstraction of the environment where the SUT is verified under one possible environment, rather than all variants.

We further assume that peers do not only respond deterministically, but also within bounded time. This restriction could be overcome by peer inspection [2]. The implementation given here works in conjunction with servers that run continuously, but could be adapted to servers that terminate after a certain number of requests have been served.

Our new cache model allows model checking of programs where data sent by a particular thread may depend on the order in which threads are executed. This is the case when threads send data that is shared among other threads in the SUT. If the content of a message depends on the thread schedule, the content will be different under some schedules. Therefore, backtracking the SUT and re-executing some operations will produce diverging results. A linear-time cache would have to abstract from this difference and ignore it. A branching-time cache can model "alternate realities" where communication data does not match previously recorded events. This allows non-deterministic SUT to be model checked using our approach. For non-deterministic peers, our approach may miss some possible peer responses, as mentioned above.

D. Example

As an example, consider a web server where the main thread accepts connections from clients and dispatches each request to a worker thread. The same page, which contains a counter, is downloaded twice. The two responses therefore differ in the value of that counter. Assume that for efficiency, request handling is not atomic (although the counter increment is). Both worker threads contain two operations:

1) Parse the GET request, retrieve and increment the counter value. The counter value is manipulated in a single atomic operation. Request parsing is thread-local, and therefore included in this step for brevity.
2) Return the response.

The order in which the requests are processed by the server is not determined. This causes the response to depend on how many requests have been served before (in total). Specifically, either the first worker thread, t_1, or the second one, t_2, may return a page containing counter value 1 or 2. Figure 2 illustrates the problem. On the left hand side, the state space exploration in the model checker is shown. GET denotes request processing, while w_1 and w_2 denote sending (writing) a response containing counter value 1 or 2.

Program execution generates two totally ordered traces. The linear-time cache model, shown in the center of Figure 2, associates one communication trace with each worker thread. States 1–10 of the state space exploration tree can be analyzed successfully using this model. Each trace contains the GET request and its response. In states 11–19, the order in which threads process the requests differ. As t_2 now processes its request first, it will send response w_1, which differs from the cached response. The linear-time cache flags this as an inconsistency and cannot continue state space exploration [2].

Our new branching-time cache model treats communication data differently. Instead of a set of linear communication traces, it contains a single tree of events. Each event sequence is cached starting from a common root node. Subsequent events are shared until communication data diverges.

In our example, the GET requests are identical and shared in the cache (see right side of Figure 2). The responses, however, differ, and are held in two different branches of the tree. Whereas the linear-time model associates a communication trace (including pointers to the request and response positions) to each connection object, the new model shares communication data between connections. However, pointers to the cache tree are still identified by connection.

Note that the branching-time cache data structure concerns only the SUT, not its peers. If such a web server with a counter is used as a peer, then its response is not consistent across requests, and hence non-deterministic. Our approach is then not applicable in general. However, our approach still allows the SUT to be model checked against one response trace of the peer.

III. Formalization of the Caching Algorithm

This formalization supersedes previous work [2] and redefines all algorithms using a branching-time model. Peer responses are required to be deterministic.

A. Stream abstraction

Communication stream data is cached as a set of immutable data elements, called *nodes*, in a mutable tree data structure. Each tree stores all observed communication traces on a given channel (see data structures in Algorithm 1). Our assignment operator $:=$ allows updates of variables and functions.

A node n is an object containing a description of its payload data, defined by function $\text{data}(n)$. Each node has either type *read* or *write*, as indicated by $\text{typeof}(n)$. Nodes are connected in a communication tree: Each node contains a possibly empty set of children, defined by $\text{childNodes}(n)$, and each node can only be the child node of one node:

$$\forall n, n_1, n_2 \cdot (n \in \text{childNodes}(n_1) \wedge n \in \text{childNodes}(n_2)) \rightarrow n_1 = n_2$$

(using reference equality). All child nodes of a given parent carry distinct data:

$$\forall n, n_1, n_2 \cdot (n_1 \in \text{childNodes}(n) \wedge n_2 \in \text{childNodes}(n) \wedge \text{data}(n_1) = \text{data}(n_2)) \rightarrow n_1 = n_2.$$

There always exists a root node *root* that carries no data. A *path* $\text{path}(n_0, n_k)$ is a sequence of nodes $\langle n_1, \ldots, n_k \rangle$ connecting a *lower node* n_k with the child n_1 of its ancestor n_0:

$$\forall i, 1 \leq i \leq k \cdot n_i \in \text{childNodes}(n_{i-1}).$$

The first node, n_0, is intentionally omitted in the path.

A *node pointer* refers to an existing node. For simplicity, we apply a Java-like notation that does not explicitly distinguish between references and actual objects. Communication of the SUT is recorded as a tree of nodes that starts at a child of the root node; $\text{path}(root, n_{curr})$ returns all communication events from the initial state to the most recent communication event n_{curr} in the current program state.

Programs operate on a set T of communication trees t, each tree having a unique root node and separate child nodes:

$$\forall t_1, t_2 \in T, \forall n_1 \in t_1, n_2 \in t_2 \cdot n_1 = n_2 \rightarrow t_1 = t_2.$$

A tree is associated to each connection object s (a socket in a given programming language). Function $\text{tree}(id)$ identifies which tree to use for a given connection. For connections to a server, identifier *id* is comprised of address and port; for distinguishing between incoming client connections, we use a global counter c, as explained in Section III-C. We assume that multiple connections to the same server peer elicit the same deterministic behavior each time, and that connections to different addresses or ports, or incoming connections, reflect different behavior on each channel. For brevity, we do not cover connections on different ports.

Finally, function $\text{response}(w) : r$ takes, for a given tree, a write pointer and returns its corresponding read pointer. It allows us to track the size and contents of each response [2].

The program state of the SUT consists of a global heap and several threads that each carry their own program counter and stack. We augment this information by the runtime data structures of our cache. The resulting *extended program state* is managed by the model checker and subject to backtracking. It includes a pair of node pointers $\langle r, w \rangle$ for each connection object s, and a global connection counter c. The extended program state is updated during backtracking, while other cache data that is not directly controlled by the model checker changes when I/O operations occur [19].

B. Example execution scenarios

In the first example (see Figure 3), a web server that returns a web page containing a counter is model checked.[1] The linear-time cache captures communication as two separate sequences (for reading and for writing). A mapping of message boundaries associates requests to responses [2]. The branching-time cache models a connection by a single tree of alternating read/write nodes (rather than two sequences). The right hand side explains how the state space shown in the tree in Figure 2 is explored. Exploration of the first half of the state space causes the two requests to be processed in the same order by the two worker threads, even though the interleavings of requests and responses differ. After half of the state space is explored, a schedule where requests are processed in reverse order is executed. Only the branching-time cache can model both outcomes.

[1] The view shown here is more detailed than in Figure 2. The linear-time cache is shown with a slightly different notation than in previous work [2], making the example easier to read for server-side verification. The read trace is shown above the written trace; the start of the written trace is in cell 2, allowing to place requests and responses in neighboring table cells.

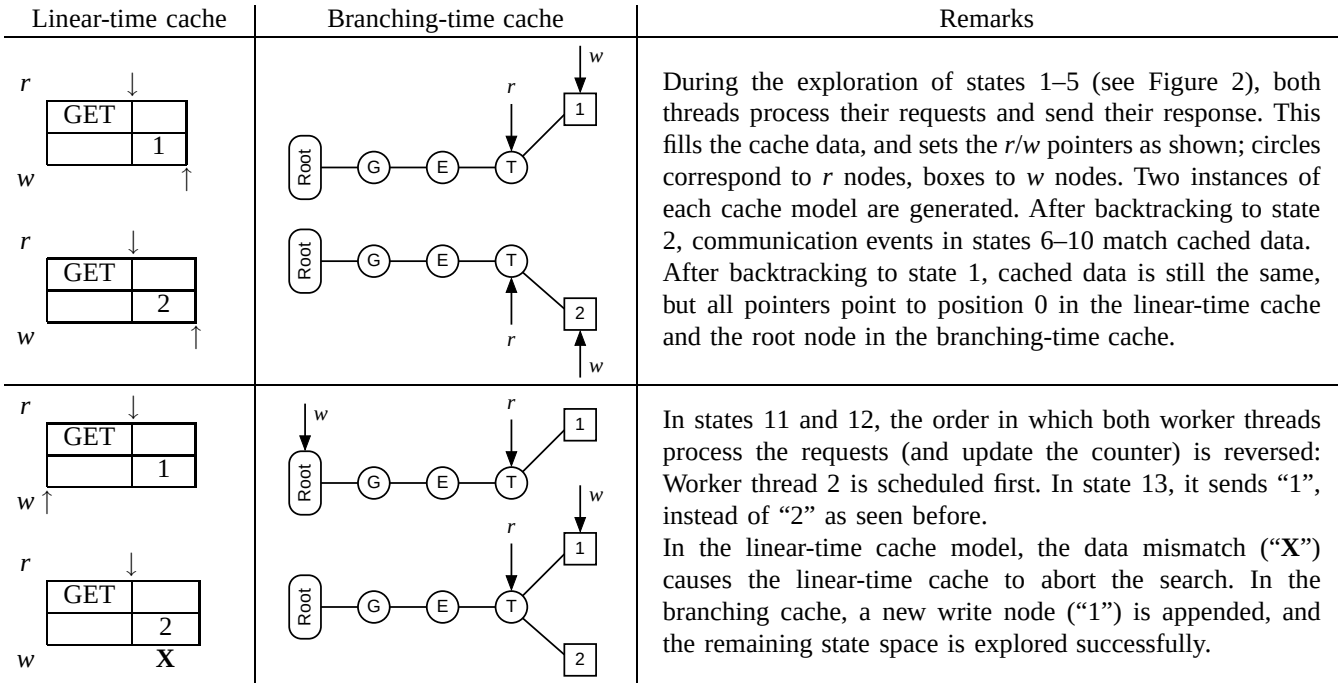

Figure 3. Example: A web server returning a page containing a counter is model checked.

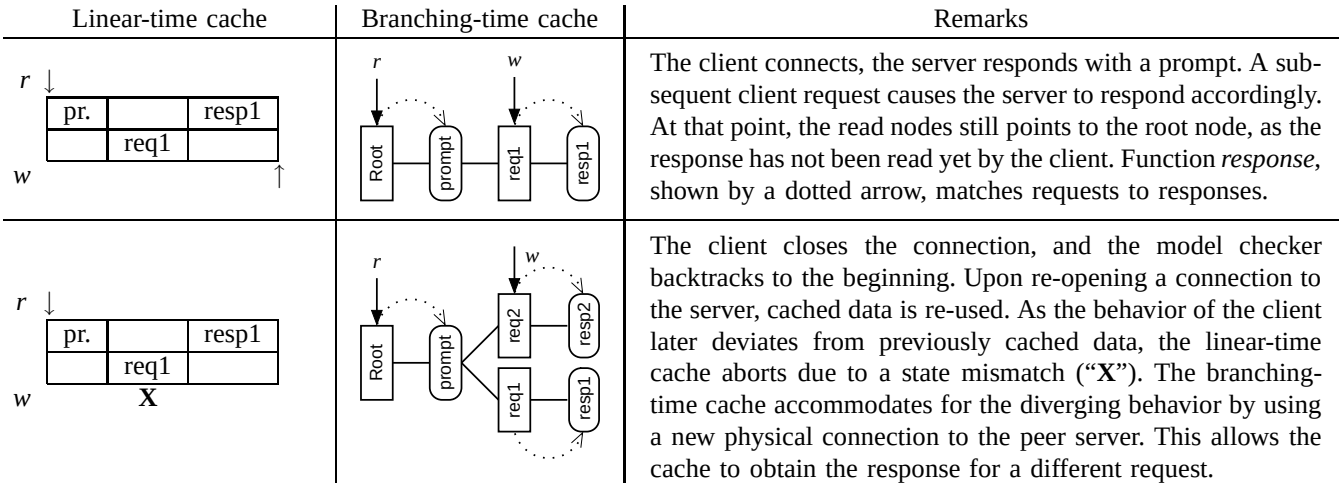

Figure 4. Example: model checking a client that non-deterministically sends two possible requests.

The second example illustrates response pointers. Here, a client is model checked. The peer (a server) initially responds with a prompt. The client then randomly sends one of two possible requests. Model checking the client will verify both of these outcomes (see Figure 4). Initially, *response* maps the root node to the initial prompt. After the client has sent its first request, that response is also cached. Assume the model checker backtracks at this point. Re-opening the connection after backtracking re-uses cached data, resetting read and write pointers to the root node. Initially, behavior matches, but as soon as the client transmits different data, it diverges from the previous observation. The branching cache handles this case correctly by using a new physical connection to retrieve the correct response for this diverging communication trace.

C. Execution semantics

Model checking of a program using our cache uses cached data when available and the underlying standard library when progressing beyond previously cached messages. The result of that library function is denoted by `lib_`*xy*(…), where *xy* represents the library function [19]. We omit handling of non-deterministic I/O failures here. Possible I/O exceptions can be modeled directly in the network library, outside the cache [18].

Without loss of generality, we assume that messages have size 1. Requests and responses may consist of multiple messages of size 1. Operations always work on a given tree t, which is omitted in the following definitions for brevity. Node pointer r denotes read data ("responses"), while node pointer w denotes written data ("requests").[2]

Function `pollResponse` (Algorithm 2) checks if a request elicits a peer response. Responses are polled proactively, to correctly treat programs where responses are processed by an independent thread. The received unread response is added as a sequence of nodes to the cache. Whenever a program reads from the same connection later on, function *response* determines the content of the cached response. It delimits the size of the response, and, by defining a path from the current read node to the end of the response, its content. Function *response* is always defined for all write nodes up to the position where a request has been cached:

$$\forall n \in \text{path}(root, w) \cdot \text{typeof}(n) = write \rightarrow \text{response}(n) \neq \emptyset.$$

It is important to note that `pollResponse` is only called for new, distinct write events. As responses are assumed to be deterministic, the response to previously cached write events is already known. Newly written data is therefore always added to the tree as a new leaf node, before `pollResponse` is called. Hence, the peer response can be added after the last write event, as a sequence of new leaf nodes. Note that the read pointer is unchanged, as the SUT itself has not seen the read data yet; instead, read data will be taken from the cache during subsequent calls to `read`. Tree data is extended by function `addNode` (Algorithm 3), which adds a new node with data d and given type (read/write) as a child node to a given parent node.

Once the model checker has performed some backtracking, write events may not match previously cached data. Therefore, function `write` behaves in two possible ways, as shown in Algorithm 4: if previously recorded communication data matches current data, then the current write node pointer is advanced accordingly. It may be necessary to skip a number of read nodes when searching for a matching write node down the tree. As peers are assumed to be deterministic, if a node has a child node which is a read node, that read node is the only child node. After all read nodes are skipped, one of the child nodes may be a matching write node. If no matching communication data exists, communication is recorded as a new branch in the communication tree, as the peer response can no longer be predicted from cached events. If the current write node is not a leaf node, a new connection to the peer has to be opened, and previously recorded data has to be replayed. Otherwise,

[2] In the case where a server is model checked, written data from the SUT is seen by the peer as a response, and data read from the peer corresponds to a previously sent request from a client. In this paper, usage of "request" and "response" corresponds to the view of the SUT.

Algorithm 1 Data structures.

map *tree* : $(integer \cup \langle address \times port \rangle) \mapsto commTree$
integer c : global connection counter
map *binding* : $socket \mapsto commTree$
communication tree t : contains immutable pointer to node *root*,
    mutable node pointers r, w, mutable map *response*
map *response* : $node \mapsto node$
object *node* : immutable object containing *data*, $type \in \{read, write\}$,
    set of node pointers *childNodes*

Algorithm 2 Function `pollResponse`.

$currNode := w$ (previously written data is new unique leaf node)
while `lib_nextByteIsAvailable` do
  $d :=$ `lib_read`$(\ldots)$
  $currNode :=$ `addNode`$(d, currNode, read)$ (add read event as new leaf node)
response$(w) := currNode$ (response of new request)

Algorithm 3 Function `addNode`(d, *currNode*, *type*).

$newNode :=$ `new Node`$(d, type)$
childNodes$(currNode) :=$ childNodes$(currNode) \cup newNode$
return *newNode*

Algorithm 4 Function `write`; $d =$ payload to be written.

while $\exists n \in$ childNodes$(w) \cdot$typeof$(n) = read$ (skip any unique read child nodes)
  $w := n$
if $\exists n \in$ childNodes$(w) \cdot$data$(n) = d$ then (data matches previously written data)
  $w := n$
  return
if childNodes$(w) \neq \emptyset$ (if w is not leaf node)
  open new physical connection for current logical connection
  replay path$(root, w)$
call `lib_write`$(\ldots, d)$
$w :=$ `addNode`$(d, w, write)$
call `pollResponse`$()$

Algorithm 5 Function `read`.

do
  $p :=$ path$(r,$response$(w))$
  $r :=$ first read node in p
while $r = \emptyset$ (if no data available, suspend current thread until update of response)
return data(r)

Algorithm 6 Function `open`.

if call is accept (incoming connection from client)
  create new socket object s
  $t :=$ tree(c)
  increment c
else (connection to server)
  $t :=$ tree$(address + port)$
if $t \neq \emptyset$ (re-use existing tree)
  $r := w :=$ *root* (of tree t)
else (first time for model checker to use this connection)
  $t :=$ new communication tree
  open new physical connection to peer
  response$(root) :=$ *root*
  $r := w :=$ *root*
  call `pollResponse`$()$ (certain protocols return data without a request)
$binding(s) := t$ (bind s to t; subsequent operations on s will access t)

a physical connection that corresponds to the current write node already exists, and is used instead.

Replaying communication works as follows: If the peer is a server accepting an infinite number of connections, a new connection is opened. For other cases (clients), a new client process is launched with the same argument as the original client. Replaying data involves sending data of each write node in the given path, and comparing received data to stored data. If data does not match, the peer is not deterministic, and model checking is aborted.

The cache proactively polls the peer process after a write event. A cached response is assumed to be complete. Function *response* defines the content of a response given the current write position (write node pointer). Available read nodes must lie on the path between the current read node and the given response. Operation `read`, shown in Algorithm 5, uses this path to ensure that the sequence is taken as a response. If the response node has been reached, no data is available, and the current thread is suspended.

A given node has either a single read node or only write nodes as children. Sequences of read nodes alternate with branching sequences of write nodes. This structure exists because read nodes are only added when a response is polled from the peer; a response is only polled when the peer has written new data, which corresponds to a leaf node in the tree. We assume that `pollResponse` retrieves a deterministic and complete response. An incomplete response still constitutes a valid scenario but omits certain schedules.

Opening a connection affects the management of active connections. Connections from a client (received via `accept`) are considered to be anonymous, while connections to a server (via `connect`) are identified by address and port number. We assume that a connection to a server on the same destination will always produce a deterministic result. Each accepted connection from a client is assumed to exhibit distinct behavior. As a shorthand for this, let a *distinct connection* be a connection to a distinct server destination, or any client-side connection. Each distinct connection is mapped to its own communication tree.

Cached data of previously recorded client communication is used after backtracking. Operation `open` re-uses the appropriate communication tree (identified by destination) if possible, and creates a new tree otherwise (see Algorithm 6). Client-side connections are re-used after backtracking in the same order in which they were created. This constitutes an abstraction in that not all possible orders in which clients can connect are accounted for. However, the model checker still explores all possible communication schedules, so the behavior of each worker thread for all possible client requests is still explored. If all clients connecting from a given destination are known to be identical, then client trees can be shared by ignoring counter c.

Operation `close` is empty. Physical connections of a communication tree are closed when that tree is no longer referenced by any unexplored states in the model checker. Keeping physical connections open allows connections to be re-used after backtracking. Therefore, closing a connection has no direct impact on cached data, but the appropriate cached branch will be re-used after backtracking, or a new one will be created, depending on data sent by the SUT.

D. Complexity

For defining complexity metrics, let the *search frontier* consist of states of which successors still have to be explored. In depth-first search (DFS), the frontier consists entirely of ancestors of the current state. In the linear-time cache [2], the number of "logical connections" (cache data instances) corresponds to the number of connections to distinct destinations. Client connections are treated as being equal. Each cache instance has a physical connection associated to it, which is closed if the SUT closes the connection. In DFS, pending connections in the search frontier are part of the current state, so the limit of the number of physical connections corresponds to the number of connections in use by the current state. For other search orders, the number of physical connections corresponds to all distinct active connections in the search frontier.

For the branching-time cache, complexity is harder to quantify. The number of logical connections (communication trees) corresponds to the number of distinct connections; not all clients are treated as equal. For identical clients, the branching-time cache may generate multiple copies of identical communication trees. In DFS, the number of physical connections in a given program state equals the number of distinct open connections. For other search strategies, the number of distinct connections in the whole search frontier may be larger, as the branching-time cache creates a new physical connection for each case where communication traces diverge. A large search frontier implies a potentially large number of physical connections.[3] After the entire state space has been explored, the number of all physical connections ever used corresponds to the number of all distinct communication traces ever generated, which is a lower bound if the entire state space of the SUT is to be explored. For a deterministic SUT, the number of physical connections corresponds to the number of logical connections.

Table I summarizes the complexity metrics. For deterministic SUT, our new branching-time cache has almost no overhead compared to the linear-time cache model. There is no overhead if clients are assumed to be identical. At the same time, for non-deterministic SUT, our branching-time cache avoids restarting peer processes and replaying previous communication data. The number of (totally or partially) distinct communication traces is typically much smaller than the number of repeated communication traces.

[3]An implementation could temporarily close these connections and re-activate them later by replaying previously transmitted data.

Table I
COMPLEXITY METRICS FOR THE LINEAR-TIME AND BRANCHING-TIME CACHE MODELS.

Metric	Linear-time cache	Branching-time cache
# logical conn. (cache data instances)		
– at a given time	# conn. to distinct dest. in all visited states	# distinct conn. in all visited states
– after full state space exploration	# conn. to distinct dest.	# distinct conn.
– for deterministic SUT	# conn. to distinct dest.	# distinct conn.
# physical connections (to peers)		
– at a given time	# conn. to distinct dest. in search frontier	# distinct comm. traces of all conn. in use in search frontier
– at a given time when using DFS	# conn. in use in current state	# distinct conn. in use in current state
– after full state space exploration	# conn. to distinct dest.	# distinct comm. traces on all conn.
– for deterministic SUT	# conn. to distinct dest.	# distinct conn.

We therefore claim that our extension of the linear-time cache model to a branching-time cache keeps the favorable performance characteristics of the original model, while enabling analysis of a larger class of systems.

IV. EXPERIMENTS

We have implemented the cache layer described here as an extension to Java PathFinder (JPF) [28]. Previous publications give an overview [2] and in-depth information [18] of our implementation. For evaluation, we compared a "no cache" setting, which obtains a new response from a fresh connection after each backtracking step, to two caching approaches: the linear-time cache [2], and our new branching-time cache. To facilitate automation, these experiments were performed on a single computer. The latest implementation of the cache is available as a JPF extension [22].

A. Example applications

Table II gives an overview of our benchmarks; some are described in detail in previous work [1], [2]. For HTTP, the protocol strings were shortened to speed up parsing. The chat server sends the input of one client back to all clients, including the one that sent the input. The original chat client transmits its ID at the beginning of each message. This ID causes a mismatch in the cached server output when the order of server worker threads is reversed after backtracking. This mismatch requires a minor manual abstraction when using the linear-time cache, and inspired this work. New applications include a WebDAV test client [17], and pws, a web server written in Java [23]. That web server supports CGI and dynamically generated page footers including the system time, which introduces non-determinism in the SUT. Logging code was removed because it caused too much overhead for model checking. In our setup, we requested a minimal page containing only the system date. This small request/response pair resulted in a smaller run than for the other HTTP server, where a larger message is processed.

Table II
EXAMPLE APPLICATIONS USED.

Application	Description
Daytime client	Sends concurrent requests to time server.
Jget, ver. 0.4.1	Multi-connection download client (simplified).
HTTP client	Test client for HTTP server (full HTTP).
HTTP server	Multi-client HTTP server (simplified for Jget).
pws ver. 0.2.3	HTTP server with CGI. Logging code removed.
Alphabet	Client requests the *n*th character from server.
Chat server	Server sends messages of one client to all clients.
WebDAV test cl.	Checks via HTTP if WebDAV server is running.
WebDAV server	HTTP server, executed using WebDAV test client.

B. Results

All experiments were run on an eight-core Intel Xeon 2.8 GHz CPU, with 16 GB of RAM, running Ubuntu 8.04.1 (Linux kernel 2.6.24) and Sun's Java VM, version 1.6.0_01. We used JPF revision 963 with 2 GB of memory, a limit that was never exhausted, and a time limit of one hour. For the linear-time cache, a newer version than the one used in previous work was utilized, to include some recent optimizations. We left diagnostic output enabled. Improvements in JPF accounted for the large reduction in the state space compared to previous experiments [2]. We verified all applications against deadlocks, uncaught exceptions, and assertion violations. Except for the WebDAV client, no program contained a critical defect that would have terminated the state space search, enabling a comparison of the full state space under different approaches.

In the WebDAV test client, we found a fault that is triggered under extremely heavy load [19]. A timer is started prematurely (in the static initializer), and may time out before it is reset later. At that point, the main thread may refer to a timer object reference that has been set to `null`. As the failure is unlikely to occur in practice, the defect has, to our knowledge, not been found before.[4]

Table III shows each application with its settings, and the results obtained when using either caching approach. The run time of the model checker, memory consumption, the number of states, and the number of bytecode instructions executed are shown. Memory usage as reported by JPF varied between runs, and was averaged over three runs. "New" states refer to distinct program states, "revisited" states to states that were visited several times after backtracking. Performance of the new branching cache is comparable to

[4]In JPF, the occurrence of a timeout event is treated non-deterministically. Both its occurrence and absence are checked, regardless of the actual time that elapsed. Timers are therefore not modeled directly, but through the non-deterministic actions that depend on timers.

Table III
RESULTS OF OUR EXPERIMENTS.

All applications are listed with the number of concurrent worker threads and the number of messages sent in each direction, per connection. Three approaches are compared: I/O managed without response caching, by restarting peer processes; and the linear-time and branching-time caches. Results include run time, maximal memory usage of JPF, the number of distinct states (in thousands), the number of states re-visited after backtracking, and the total number of instructions executed (in thousands) during state space exploration.

App.	#	#	No cache					Linear-time cache					Branching-time cache				
	th.	msg.	Time [m:ss]	Mem. [MB]	States [10^3] new	States [10^3] revis.	Instr. [10^3]	Time [m:ss]	Mem. [MB]	States [10^3] new	States [10^3] revis.	Instr. [10^3]	Time [m:ss]	Mem. [MB]	States [10^3] new	States [10^3] revis.	Instr. [10^3]
Daytime client	2	1	0:27	20	0.3	0.1	51	0:01	21	0.3	0.1	51	0:01	21	0.3	0.1	71
	3	1	> 1 h					0:25	190	55.2	36.4	9593	0:36	224	55.2	36.3	21332
HTTP c.	2	1	3:01	69	0.9	0.7	115	0:10	185	0.9	0.8	116	0:12	222	0.9	0.8	115
HTTP s.	2	1	> 1 h					4:30	158	403.2	392.0	142529	1:47	245	132.1	130.5	19436
HTTP s./ctr	2	1	> 1 h					Cannot be handled due to state mismatch.					2:13	246	137.0	135.4	21010
pws	2	1	> 1 h					Cannot be handled due to state mismatch.					1:43	276	70.6	57.2	38094
Jget	2	1	13:03	155	4.8	4.8	305	0:15	133	5.2	5.2	383	0:16	247	4.8	4.8	305
Alphabet client	2	1	1:10	21	0.3	0.5	26	0:01	22	0.3	0.5	25	0:01	22	0.3	0.5	26
		2	5:29	34	1.1	2.5	140	0:02	33	1.1	2.6	136	0:02	33	1.1	2.5	140
		3	14:56	33	2.8	6.7	410	0:03	43	3.0	7.2	409	0:03	44	2.8	6.7	410
		4	31:59	36	5.8	14.4	917	0:06	53	6.2	15.5	925	0:05	54	5.8	14.4	917
		5	59:32	34	10.4	26.8	1761	0:09	69	11.2	28.9	1780	0:09	72	10.5	26.8	1761
	3	1	> 1 h					0:06	45	6.1	21.8	764	0:06	52	6.1	21.8	791
		2						0:40	100	48.5	194.8	8953	0:37	142	44.7	179.3	8801
		3						2:39	113	189.7	801.1	40165	2:20	233	167.7	703.5	37510
		4						7:51	160	537.4	2330.1	122731	6:44	242	471.1	2028.3	113368
		5						19:37	339	1267.6	5594.0	304339	16:50	369	1113.7	4883.4	282032
	4	1	> 1 h					2:16	134	137.8	725.1	23351	2:12	242	137.8	725.1	24135
		2						35:22	411	1948.4	11261.1	493525	30:55	478	1691.8	9741.6	457694
Alphabet server	2	1	2:22	38	0.0	0.0	8	0:02	16	0.0	0.0	6	0:05	16	0.0	0.0	8
		2	4:56	44	0.1	0.1	14	0:02	16	0.1	0.1	12	0:05	18	0.1	0.1	14
		3	8:17	39	0.2	0.2	24	0:03	16	0.1	0.2	19	0:05	20	0.2	0.2	24
		4	12:25	45	0.2	0.2	35	0:03	16	0.2	0.2	29	0:05	20	0.2	0.2	35
		5	17:24	44	0.3	0.3	50	0:03	16	0.3	0.3	40	0:06	20	0.3	0.3	49
	3	1	48:49	43	0.4	0.7	64	0:03	16	0.3	0.6	47	0:07	30	0.4	0.7	64
		2	> 1 h					0:04	29	1.1	2.2	169	0:08	32	1.1	2.2	208
		3						0:06	47	2.4	4.7	382	0:10	49	2.3	4.6	472
		4						0:06	59	4.3	8.6	719	0:11	67	4.2	8.4	893
		5						0:08	63	7.0	14.0	1211	0:13	68	7.0	13.9	1506
	4	1	> 1 h					0:10	51	4.8	14.5	1025	0:16	70	7.5	22.0	1825
		2						0:21	60	20.9	62.7	4449	0:30	78	26.2	76.4	6944
		3						0:45	95	49.8	149.4	11179	1:03	75	66.2	192.3	18330
		4						1:24	113	100.9	302.5	23359	2:03	155	139.3	404.0	39722
		5						2:23	118	183.0	549.0	43300	3:44	214	260.2	753.5	75739
	5	1	> 1 h					0:36	59	36.3	145.0	9625	1:00	94	52.4	205.7	16394
		2						3:19	96	235.8	942.9	62481	3:53	225	228.4	900.6	78032
		3						9:45	200	684.0	2735.7	191300	11:22	347	678.9	2682.5	241945
		4						23:47	429	1632.7	6530.4	471592	27:35	513	1637.6	6480.2	599098
		5						54:18	545	3415.0	13659.7	1007965	58:42	845	3442.8	13638.9	1282426
	6	1	> 1 h					4:20	218	261.8	1308.9	83824	8:01	260	418.0	2068.4	157405
		2						42:43	543	2541.9	12709.0	807192	47:04	662	2360.6	11728.5	968774
	7	1	> 1 h					36:35	493	1832.7	10995.7	687803	> 1 h				
Chat server, abstracted	2	1	> 1 h					0:03	20	1.1	1.2	38	0:06	29	1.5	1.6	55
	3	1						0:12	57	24.2	44.0	815	0:24	58	46.5	81.1	1716
	4	1						8:20	234	1116.4	2846.7	50558	20:13	585	2718.0	6748.1	130546
Chat server, full	2	1	> 1 h					Cannot be handled due to state mismatch.					0:10	29	1.5	1.6	55
	3	1											0:57	55	46.5	81.1	1716
	4	1											24:25	595	2718.0	6748.1	130546
WD. c.	2	1	3:50	74	3.4	2.8	319	0:04	74	3.4	2.8	306	0:04	74	3.4	2.8	306
WD. s.	2	1	> 1 h					13:56	249	1446.2	1443.5	115493	13:30	348	1446.2	1443.5	113992

Table IV
CACHE USAGE.

The number of times a connection had to be restarted after backtracking when no response caching was used, is compared to cache hits/misses and number of peer restarts for the branching-time cache. Without response caching, peers have to be restarted whenever the model checker backtracks the SUT; cached requests are then replayed up to the new current state of the SUT. With response caching, most requests match previously recorded data, allowing the response to be read from the cache. Note that a single transition between states in JPF may include multiple read/write operations of the same thread; hence, the number of cache hits may be larger than the number of states reported by JPF. Restarts/connection counts include the initial connection.

App.	# th.	# msg.	No cache	Branching-time cache		
			# (re)starts/ (re)conn.	# hits	# misses	# (re)starts/ (re)conn.
Daytime client	2	1	1056	10710	60	2
	3	1	> 1 h	4788180	90	3
HTTP c.	2	1	4835	95940	533	2
HTTP s.	2	1	> 1 h	108732	142	2
HTTP s./ctr	2	1	> 1 h	109584	142	4
pws	2	1	> 1 h	4561	18	3
Jget	2	1	28701	3284	124	3
Alphabet client	2	1	2644	254	2	2
		2	12142	1318	4	2
		3	32572	3698	6	2
		4	69172	8098	8	2
		5	128232	15370	10	2
	3	1	> 1 h	9195	3	3
		2		80115	6	3
		3		326166	9	3
		4		970551	12	3
		5		2397147	15	3
	4	1	> 1 h	280372	4	4
		2		3991996	8	4
Alphabet server	2	1	268	14	2	2
		2	560	46	4	2
		3	940	94	6	2
		4	1408	158	8	2
		5	1972	238	10	2
	3	1	5559	232	3	3
		2	> 1 h	953	6	3
		3		2358	9	3
		4		4663	12	3
		5		8084	15	3
	4	1	> 1 h	4795	4	4
		2		23518	8	4
		3		68601	12	4
		4		157036	16	4
		5		310039	20	4
	5	1	> 1 h	52442	5	5
		2		311092	10	5
		3		1065266	15	5
		4		2799980	20	5
		5		6240634	25	5
	6	1	> 1 h	543719	6	6
		2		4039171	12	6
Chat server, abstracted	2	1	> 1 h	229	2	2
	3	1		11207	3	3
	4	1		873089	4	4
Chat server, full	2	1	> 1 h	229	4	4
	3	1		11207	18	18
	4	1		873089	96	96
WD. c.	2	1	8907	0	0	0
WD. s.	2	1	> 1 h	1900166	870	2

the old linear cache. At the same time, our cache can handle complex programs such as the unabstracted chat server, and HTTP servers returning dynamic information such as page counts or the current time.

Table IV shows the effectiveness of our response cache. Without caching, peer processes have to be restarted each time when backtracking occurs. Restarting and replaying previous communication causes an extreme slowdown.

Previous work shows that model checking with our I/O caching approach is orders of magnitudes faster than model checking using centralization [2]. By changing the cache model from linear to branching time, we can handle a large number of programs than before, while keeping the performance advantage of caching.

V. RELATED WORK

Software model checkers [5], [7], [10], [13], [15], [16], [28] backtrack a program to analyze various outcomes of each non-deterministic decision in the SUT. Backtracking may cause a repetition of previously executed operations. This requires special treatment of communication operations, as described in the introduction, and more in depth in a recent survey [3]. Peer processes may be included in the resulting system [11], [20] or modeled by a (possibly abstracted) environment process or stub [7], [10], [14]. Generation of the optimal abstraction can be automated [7], [10]. Our approach requires no stubs but assumes deterministic peer behavior. Soundness is lost if a peer is non-deterministic [4].

A general solution to model checking multiple communicating processes is to lift the power of a model checker from a process to operating system (OS) level, treating the entire OS and all processes running within as the SUT. This way, the effect of I/O operations are fully visible inside the model checker, and the combined state space of all processes is explored. An existing system that stores and restores full OS states is based on user-mode Linux [21]. Compared to that work, our approach analyzes a single process at a time inside a model checker, while running other processes normally, making it more scalable than such multi-process model checking approaches.[5]

External processes can also be backtracked in tandem with the system under test, for instance, by restarting them [11], [16]. In existing implementations, one central scheduler controls and backtracks several processes, effectively implementing a multi-process model checker [16], [20]. Like all approaches controlling multiple processes inside the model checker, it incurs a massive state space explosion. In our approach, restarting communication to peers corresponds to

[5]It seems possible to implement a multi-process model checker that executes all thread schedules for only one process, while executing only one schedule for all other processes. This would achieve a similar result as our I/O cache, but has, to our knowledge, not been implemented so far.

backtracking of peers, but our cache eliminates backtracking of peers whenever communication data matches.

Finally, the application of a corresponding program transformation [25] also allows multiple processes to be model checked in a single-process model checker. In that approach, I/O is modeled in the transformed program and fully controlled by the model checker [1], [6]. Recent work has implemented this approach in a similar way, but sacrificed full automation in favor of manual instrumentation of communication operations [8]. That tool has another mode in which it can run, replacing peer processes with stubs that replay previously recorded communication [8].[6] At a high level, a stub that models a previously recorded trace works like our cache. Data of the real network is mimicked by a program (stub) as opposed to a data structure (cache). Like in our approach, stubs assume that peer processes are deterministic. In addition to that, for stubs, communication of the SUT itself is not allowed to diverge from the behavior the stub was recorded or written against. This is because the functionality of a stub is fixed prior to execution. Our approach does not require a deterministic SUT, and eliminates the need of an intermediary stub program. Furthermore, it even allows model checking of applications where external processes are not running on a platform that the model checker supports.

Our approach supersedes previous work, which used a linear-time cache. The linear-time cache requires a high degree of determinism in the SUT [2]. Using a communication model that allows for branching, alternative communication traces, we can eliminate this restriction. We regard peers as deterministic to allow the utilization of caching, but we believe that this achieves an ideal trade-off between exhaustive coverage of all systems, and scalability. In the original linear cache [2], any divergence of the SUT behavior causes the cache to abort. It is conceivable that the peer be restarted and the cache be cleared in that case, discarding any previous data. However, this would negate much of the performance advantage of caching, which is why we developed the branching-time cache.

The way the linear-time and branching-time caches deal with communication data resembles the relation between linear-time logic and branching-time logic [9], [24]. Even though we do not model check temporal logic formulae, our cache structure can deal with alternate sequences of events in a way that is similar to how existential path quantifiers in branching-time logic operate.[7]

[6]The recorder has not been released yet. We attempted to use NetStub on a manually adapted chat server, but a defect in JPF thwarted our efforts.

[7]By default JPF checks against assertion violations, unchecked exceptions, and deadlocks. Temporal logic properties can be implemented as property listeners, and verified in conjunction with other extensions such as our network layer [28].

VI. Conclusions and Future Work

When model checking communicating programs, processes outside the model checker are affected by communication but not subject to backtracking. Because backtracking is not applicable to external communication, input/output operations cannot be executed directly. Their effects can sometimes be subsumed by stubs; alternatively, multiple processes can be executed inside the model checker. The former is difficult to automate, while the latter suffers from scalability problems.

We defined a caching semantics for network communication between the model checker and peer processes. Whenever communication exceeds previously stored data, or diverges from it, the cache transmits data physically over the network. Only when necessary, new connections are created and previously recorded traces are replayed. This paper introduces a branching-time semantics for the cache, which allows non-deterministic programs to be model checked. For deterministic programs, our enhanced model delivers similarly fast performance when compared to its predecessor. Unlike most previous work, we can handle implementations using standard network libraries without any manual intervention, while eliminating some scalability issues of some related approaches. We also have a fully working and very scalable implementation of our algorithm for the Java PathFinder model checker, which we applied to several applications. During these experiments, we have found a defect in a widely used application.

Current work focuses on model checking applications communicating by TCP. This protocol is reliable in the sense that message order is preserved, and messages are not lost. However, we think that the concept can be modified and be applied to unreliable protocols such as the User Datagram Protocol (UDP). Such a modification should also allow us to cover non-deterministic responses. We will also work on the issue of slow responses, by implementing a tool that inspects state of peer applications. Finally, it remains to be seen how far our approach, which has so far been tried on service-oriented client-server systems, is applicable to peer-to-peer systems or multicast protocols.

Acknowledgments: We thank the reviewers for their helpful comments. This work was supported by a *kakenhi* grant (2030006) from JSPS.

References

[1] 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.

[2] 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.

[3] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Tools and techniques for model checking networked programs. In *Proc. 9th Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2008)*, Phuket, Thailand, 2008. IEEE.

[4] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, M. Yamamoto, and K. Takahashi. Verification of multi-process systems by combining model checking with testing. 2009. To be published.

[5] 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.

[6] 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)*, pages 219–228, Shanghai, China, 2007. IEEE Computer Society.

[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] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *Proc. 8th ACM Symposium on Principles of Programming Languages (POPL 1981)*, pages 164–176, Williamsburg, USA, 1981. ACM.

[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 *Proc. 24th Int. Conf. on Software Engineering (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 *Proc. 22nd Int. Conf. on Software Engineering (ICSE 2000)*, pages 439–448, Limerick, Ireland, 2000. ACM Press.

[14] 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.

[15] 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.

[16] 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.

[17] B.C. Holmes. A Simple PROPFIND/PROPPATCH Client, 2000. http://www.bcholmes.org/geek/slide-my-webdav-client.html.

[18] W. Leungwattanakit. Networked software model checking by extending Java PathFinder. Master's thesis, University of Tokyo, 2008.

[19] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto. Verifying networked programs using a model checker extension. In *Proc. Int. Conf. on Software Engineering (ICSE 2009), Companion Volume*, pages 409–410, Vancouver, Canada, 2009. ACM Press.

[20] 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.

[21] 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.

[22] NASA. Java PathFinder, 2009. http://javapathfinder.sourceforge.net/.

[23] Pegasi LUG. Pegasi Web Server, 2005. http://sourceforge.net/projects/pws.

[24] 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.

[25] 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.

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

[27] A. Tanenbaum. *Computer Networks*. Prentice-Hall, 2002.

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