Modular Software Model Checking for Distributed Systems

Watcharin Leungwattanakit, Cyrille Artho, Masami Hagiya,
Yoshinori Tanabe, Mitsuharu Yamamoto, and Koichi Takahashi

Abstract—Distributed systems are complex, being usually composed of several subsystems running in parallel. Concurrent execution and inter-process communication in these systems are prone to errors that are difficult to detect by traditional testing, which does not cover every possible program execution. Unlike testing, model checking can detect such faults in a concurrent system by exploring every possible state of the system. However, most model-checking techniques require that a system be described in a modeling language. Although this simplifies verification, faults may be introduced in the implementation. Recently, some model checkers verify program code at runtime but tend to be limited to stand-alone programs. This article proposes cache-based model checking, which relaxes this limitation to some extent by verifying one process at a time and running other processes in another execution environment. This approach has been implemented as an extension of Java PathFinder, a Java model checker. It is a scalable and promising technique to handle distributed systems. To support a larger class of distributed systems, a checkpointing tool is also integrated into the verification system. Experimental results on various distributed systems show the capability and scalability of cache-based model checking.

Index Terms—Software model checking, software verification, distributed systems, checkpointing.

1 Introduction

DISTRIBUTED systems [1] are complex due to multiple units of execution operating in parallel. They are composed of several processes, generally running on different platforms. Processes communicate with each other over a network. As network communication is not perfect in practice, messages may be delayed or even lost. However, well-written applications should be able to continue working as expected or terminate gracefully, even encountering such unpredictable circumstances. Furthermore, a process may be *multithreaded* by creating dedicated threads that handle connections with other processes [2]. The threads in the process are interleaved according to the policy adopted by the operating system. How the threads will be interleaved is usually not known beforehand.

Faults caused by concurrency are usually difficult to detect and reproduce since they only happen on a certain timing. *Testing* [3] does not cover every possible way that a system can be executed, even if it runs the system several times. Concurrent faults therefore may still be remaining, even after the application has passed a rigorous test suite. *Model checking* [4] is a technique to detect property violations in a concurrent system by exploring every possible execution path. Accordingly, every possible *state* of the system is checked against given properties. This technique is especially useful for quality assurance of safety-critical systems and core algorithms/protocols of large systems. Model checking was originally developed for hardware verification, but the concept of state-space exploration has been applied to a wide range of software systems as well [5], [6].

In the traditional approach, a system to be verified is abstracted into an input language supported by the model checker. The model checker then generates a directed graph that represents a state space of the system. It traverses the graph and checks if the desired properties hold at every state. This activity is referred to as *state exploration*. After verification of the model, the system is usually implemented in a programming language, based on the verified model.

Although model checking originally required an abstract model, recent work in the community applies model checking directly to an implementation; this activity is often called *software model checking*. Direct verification on real code gives more confidence on software safety. The fact that system design meets a specification does not imply that the implementation does as well. Many concurrency-related faults are introduced by programming mistakes during the implementation phase, such as race conditions, deadlocks, etc. Verification in the design phase cannot guarantee the absence of faults in the final deliverables. The software model-checking community focuses on the analysis of real implementations, written in mainstream programming languages such as Java and C [6], [7]. In the software development life cycle, software verification is not intended to

- *W. Leungwattanakit and M. Yamamoto are with the Department of Mathematics and Informatics, Chiba University, Chiba, Japan.*
- *C. Artho and K. Takahashi are with the Research Institute for Secure Systems at the National Institute of Advanced Industrial Science and Technology (AIST), Amagasaki and Tsukuba, Japan.*
- *M. Hagiya is with the Department of Computer Science, University of Tokyo, Tokyo, Japan.*
- *Y. Tanabe is with National Institute of Informatics (NII), Tokyo, Japan.*

Manuscript received April XX, 2013; revised September 23, 2013.

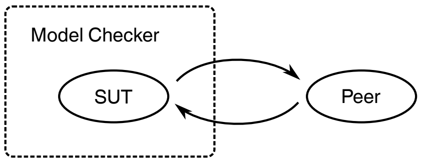


Fig. 1. Only one process is verified by a model checker.

replace system design verification; instead, it serves as the last line of defense before a software release. Careful verification on both the system design and the system implementation is important in software development.

Most software model checkers only verify a single-process program. Network communication is not modeled in such model checkers. Extending the capability of existing model checkers to support network communication is not straightforward. Each process in a system may interleave with each other, generating an extremely large composite state space. Furthermore, model checking on real code explores many more states than model checking on modeling languages because of the more detailed scenario. Full state exploration therefore is not scalable. To improve scalability, one can separate the processes in a system into a *system under test* (*SUT*) and *peer processes* [8]. The SUT is a process that a tester wants to verify in a software model checker. The peer processes are part of the system that runs as normal. They interact with the SUT and are necessary for the system to execute. The model checker only verifies the SUT while the peer processes are not subject to verification, as shown in Figure 1. The peer processes are assumed correct (and can be verified in a separate verification run). Although this approach trades accuracy for scalability, we believe it is a reasonable trade-off. Developers usually only want to test the process they develop, not the environment, i. e., peer processes.

Compared to approaches that use stubs to emulate peer behavior [9], our approach is fully automated and guaranteed to be accurate, as the real peer process is used. Compared to approaches that transform peer processes into threads (executed inside the SUT) [10], [11], our approach is more scalable because the state space of only a single process is fully analyzed [12].

When a model checker reaches a terminal state during state-space exploration, it reverts to a previously observed state that still has unexplored successor states. This activity is called *backtracking* or *rollback.* In distributed systems, backtracking poses a challenge in state synchronization. Network operations, once performed, cannot be reverted since the environment (peers) has been affected. Peers that are not rolled back cannot correctly interact with the SUT when state space exploration continues after backtracking.

This article presents the concept of *cache-based model checking*, which concentrates on verifying a single process in a distributed system [8], [12], [13]. A series of approaches, based on this concept, is explained to support a model checker in dealing with distributed applications. Cache-based model checking emulates the behavior of the environment by recording the communication between the SUT and the environment. Communication messages are stored in a cache that interoperates with the model checker. The cache adapts itself according to changes of the SUT state so that it can interact with the SUT on behalf of the real peer processes. Cache-based model checking offers a scalable method to verify a single process in a system since the state space of one process is explored rather than the composite state space of all processes. It also has the benefit of allowing testers to verify a system with a peer process that is difficult to model. Furthermore, *process checkpointing* [14] is introduced into software model checking to capture the environment state under certain circumstances where the cache is not sufficient to emulate the environment behavior. The conditions and assumptions required to use each approach are discussed in this article. Our experiment both applies the pure cache-based model checking approach and cache-based model checking with the process-checkpointing function. It shows that cache-based model checking with process-checkpointing support, given appropriate optimization, is more powerful than, yet as scalable as pure cache-based model checking.

Our contributions are as follows: This article

1) proposes methods to synchronize a process controlled by a software model checker with external processes;
2) formalizes and classifies applications by I/O determinism;
3) presents cache-based model checking, which improves the performance of single-process verification;
4) integrates checkpointing into software model checking to overcome the limitation of caching;
5) evaluates the performance and applicability of each approach;
6) introduces the concept of *trace convergence*, which prevents a single-process model checker from discovering every fault in a program.

This article is organized as follows. Section 2 reviews the background and fundamental knowledge required to read the article. Section 3 explains why state synchronization is necessary in verification of multi-process systems. Section 4 presents the concept of cache-based model checking. Section 5 shows how to implement the concept on *Java PathFinder*, a Java model checker. Section 6 explains why and how we use a checkpointing tool in the verification of distributed systems. Section 7 shows and discusses experimental results. Section 8 covers related work in software model checking. Section 9 summarizes the article and concludes.

2 Preliminaries

This section reviews software verification techniques and fundamental knowledge. The concept of model checking is briefly explained. The definition of transition systems with input/output is reviewed since it is used to formalize software processes in later sections.

2.1 Model Checking

Model checking is a technique for concurrency analysis. It systematically explores the entire state space of a system by analyzing the outcome of all possible traces in a system, starting from a given initial state. A system to be analyzed is usually represented by a transition system such as a *Kripke structure* [4].

The system specification defines a set of properties that must hold for a particular system. Model checking verifies whether the specification holds. A tool that performs this verification is called a *model checker*. When analyzing software, the specification also includes the absence of assertion violations and uncaught exceptions.

Model checking was originally developed to verify hardware systems, algorithms, and protocols. Because of its success, model checking has later also been applied to software systems. "Classical" model checkers take a system description in a special-purpose modeling language as input. A developer may write an algorithm in the modeling language during the design phase and verify it by a model checker.

Modern software model checkers have been designed to accept the actual implementation of programs, written in standard programming languages. These model checkers either operate directly on the program or convert a program to an abstract representation that is then taken as input by the model checker. This kind of direct verification is more precise since it can detect human errors introduced during the implementation phase. Examples of such model checkers include *SLAM* [15], *Blast* [5], and Java PathFinder [6]. Java PathFinder is used as the base model checker to implement the approaches presented in this article.

A software model checker exhaustively explores, by a search algorithm, the state space of a target program to find an *error state*, which violates specifications. If such a state is found, the model checker will report an *error trace*, a path from the initial state to the error state as evidence. The error trace serves as a counterexample that proves incorrectness of the system. Programmers then can make use of this information to analyze the fault. If all states are visited and no error state is found, the program meets the specifications. Software model checking surpasses software testing in terms of coverage since it takes every possible non-deterministic outcome into account. Since this article only concentrates on software model checking, we simply refer to software model checking as "model checking" for the rest of the article unless stated otherwise.

2.2 Input-output Transition Systems

A labeled transition system with input/output is a mathematical object that can be used to formalize the behavior of communication systems [16]. We use this representation to classify the behaviors of different types of systems.

A labeled transition system with input/output, abbreviated to LTS, is a 5-tuple $\langle S, L_I, L_O, T, s_0 \rangle$, where

- S is a finite set of *process states;*
- L_I and L_O are a set of *input labels* and a set of *output labels*, respectively;
- $L_I \cap L_O = \phi$.
- $T \subseteq S \times (L_I \cup L_O \cup \{\tau\}) \times S$ is a set of *transitions;*
- $\tau \notin L_I \cup L_O$ is an *unobservable action;*
- $s_0 \in S$ is the *initial state;*

A process moves from one state to another state according to a label. The label is an action of the process and can be either input from another process, output to another process, or an unobservable action. The unobservable action can be considered an internal computation that does not involve communication with the environment. Although there can be a number of different unobservable actions, this formalization uses label τ to represent all of them, without distinction. In contrast with unobservable actions, we call action $l \in L_I \cup L_O$ an *observable action*. The set of labels of LTS P, denoted by $L(P)$, includes every possible label, that is $L_I \cup L_O \cup \{\tau\}$.

2.3 Properties of Verification Systems

Error states in an LTS represent undesirable situations in a concrete system such as deadlocks, assertion failures, or unhandled exceptions. The objective of model checking is to find such error states. Model checkers return a verdict on the presence of system faults after exploring the state space. In this work, only the state space of the SUT is taken into account; external processes are not covered. If the model checker rejects a system that contains no defect, its report is called a *false positive*. False positives may arise due to the imprecision of the LTS, which shows a behavior that never happens in the real system. We also say the report is a *true positive* if an error state really exists. Conversely, if the model checker accepts a defective system, its report is called a *false negative*. In this case, the model checker misses a defect in the system, because the LTS is too coarse to capture every system behavior. Similarly, a *true negative* means the system does not contain a defect. Both false positives and false negatives are properties used in describing a verification system.

3 State Synchronization

Model checking involves system states and transitions. In a software system, a state can be composed of stack data, heap data, and thread information. The software state changes by instruction execution, including communication with other systems.

Our approach maintains scalability by verifying a single process of a distributed application in a model checker. This kind of verification is called *single-process* *verification*. In single-process verification, both an SUT and a peer change their states during execution (verification). However, the SUT state can be reverted by a model checker while the peer state can only move forward by executing program instructions. This restriction leads to an inconsistency between both processes during verification, because the peer is not aware of the model checker. The definition of consistency is formally defined in Section 3.3. The states of both processes must be synchronized with each other to preserve consistency of the system behavior. This section discusses the problems related to state synchronization and possible solutions.

3.1 Definitions and Terminology

To describe our approach, processes in a verification system are modeled by a labeled transition systems (LTS) with input/output (see Section 2.2). The state of the process changes during execution. Instruction execution is modeled as a transition. The transition from state s to s' by action l is denoted by $s \xrightarrow{l} s'$. An action can be either reading input, writing output, or an unobservable action. This notation is extended for multiple actions. If $s \xrightarrow{l_1} s'$ and $s' \xrightarrow{l_2} s''$, we write shorthand notation $s \xrightarrow{l_1, l_2} s''$. This shorthand notation is used to describe reachablity. Let $P \langle S, L_I, L_O, T, s_0 \rangle$ be an LTS; state $s \in S$ is *reachable* iff there exist actions $l_1, \ldots, l_n \in L(P)$ such that $s_0 \xrightarrow{l_1, \ldots, l_n} s$. A system in the initial state can be in a reachable state by performing available actions. Generally, we only consider reachable states and assume every state in S is reachable for the rest of the article.

A system reaches any reachable state by taking a sequence of actions, resulting in an *event trace*. In the example above, $\langle l_1, \ldots, l_n \rangle$ is an event trace from state s_0 to state s. An event trace may include a number of unobservable actions. When we consider communication between processes, taking only observable actions into account is more convenient. We therefore define *observable event traces*, which exclude action τ from consideration. Let s and s' be states, $m, m_i \in L_I \cup L_O$;

$$\begin{array}{lcl} s \overset{\epsilon}{\Rightarrow} s' & \Leftrightarrow & s = s' \text{ or } s \xrightarrow{\tau, \ldots, \tau} s' \\ s \overset{m}{\Rightarrow} s' & \Leftrightarrow & \exists s_1, s_2 : s \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{m} s_2 \overset{\epsilon}{\Rightarrow} s' \\ s \overset{m_1, \ldots, m_n}{\Longrightarrow} s' & \Leftrightarrow & \exists s_1, \ldots, s_n : s \overset{m_1}{\Rightarrow} s_1 \overset{m_2}{\Rightarrow} \ldots \overset{m_n}{\Rightarrow} s_n = s' \end{array}$$

We use a double arrow to represent the observable actions observed while a system is moving from state s to s'. In the first form of the double-arrow notation, only τ-transitions are taken by the system to move from state s to s', so no event trace is observed. This empty event trace is denoted by ϵ. In the second form, there is exactly one transition associated with a message in a path from state s to s'. Other transitions in the path are not observed. The observable event trace, therefore, contains one observable action m. In the third form, multiple messages are observed along a path from state s to s'. In this paper, an observable event trace is denoted by $\langle m_1, \ldots, m_n \rangle$. For simplicity, we assume every LTS reads and writes messages in an alternating way. Each reading and writing operation is performed in one transition. Formally, every observable event trace $\langle m_1, \ldots, m_n \rangle$ satisfies $\forall_{1 \le i < n} : (m_i \in L_I \rightarrow m_{i+1} \in L_O) \wedge (m_i \in L_O \rightarrow m_{i+1} \in L_I)$. This assumption is only for this discussion of our approach. Our implementation supports partial reads and writes, i. e., multiple read or write actions may be performed consecutively.

The same event trace does not necessarily lead to the same state due to *non-determinism* in the transition system. For example, both $s \xrightarrow{l} s'$ and $s \xrightarrow{l} s''$, where $s' \neq s''$, hold at the same time if both (s, l, s') and (s, l, s'') are in the set of transitions. Determinism of an LTS, however, is determined differently in this article. We only consider output traces of a system, given an input trace. Unobservable actions are ignored. Under this definition, if an LTS is to perform a write action, only one write action will be available at a given state. This is formally defined as follows.

$$\begin{array}{l} \forall k \cdot \left(\forall j < k \cdot \left(m_j = m'_j\right)\right) \\ \wedge \left(m_k \in L_O \vee m'_k \in L_O\right) \rightarrow m_k = m'_k \end{array} \tag{1}$$

where $\langle m_1, \ldots, m_k \rangle$ and $\langle m'_1, \ldots, m'_k \rangle$ are observable event traces starting from state s_0; k is the length of both observable event traces. To distinguish our definition from classical determinism, we call it *I/O determinism.* This definition is very important in the classification of applications introduced later. Some systems produce deterministic output, although they contain internal non-determinism due to concurrency. I/O determinism is usually desirable since it shows that the output of the system is independent of the thread schedule. Note that this definition of determinism does not correspond to a *stateless* system. If a system were completely stateless, then the output symbol would only depend on the last input symbol, regardless of any prior events.

3.2 Classification of Applications

In this article, distributed applications are classified into a number of classes according to their I/O determinism. For simplicity, we assume a distributed application is composed of an SUT and a peer process. Both processes may be either deterministic or non-deterministic. Therefore, applications can be classified into four groups, as shown in Figure 2.

An application may be I/O non-deterministic because of unpredictable external data such as random numbers, file contents, system time, etc. Such data is possibly different in each run, so the computation does not give the same result (output) on each input.

3.3 State Consistency

As mentioned before, the state of an SUT running inside a model checker may be inconsistent with the state of a peer during verification. This happens when the model checker backtracks the SUT to explore an alternative path. Two problems arise:

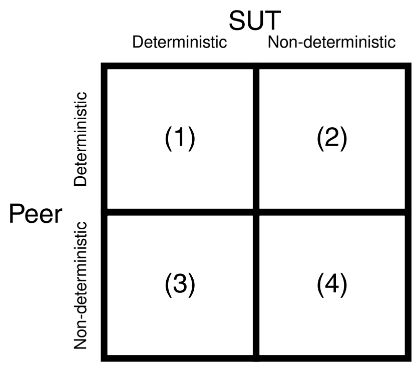


Fig. 2. Four groups of target systems classified by I/O determinism of SUT and peers.

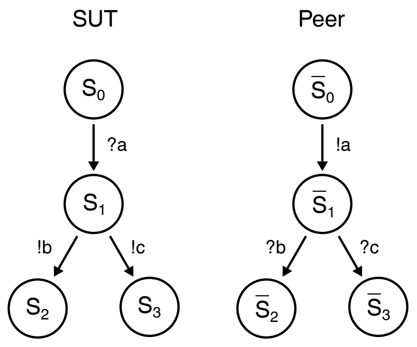


Fig. 3. The state spaces of the processes in a system. Each state of the SUT corresponds to a unique peer state. $con(s_i, \bar{s}_i), i \in \{0, 1, 2, 3\}$.

1) The SUT may send a duplicate output to the peer that was already sent before backtracking. The peer, which does not expect the duplicate message, may deal with that input in a wrong way.
2) The SUT may expect an input from the peer it has received before backtracking. Since the peer is not aware of backtracking, it does not send the same message again.

Both problems are caused by an inconsistency between the states of the SUT and the peer. A formal definition of state consistency allows a classification of different approaches to deal with this problem. Without loss of generality, we assume a single peer process for the remainder of this discussion.

Both an SUT and a peer are modeled as LTS. Let the SUT and peer be represented by LTS $P\left\langle S, L_I, L_O, T, s_0\right\rangle$ and $Q\left\langle \bar{S}, L_O, L_I, \bar{T}, \bar{s_0}\right\rangle$, respectively. The labels in L_I represent every possible *message content* process P may read while the labels in L_O are message contents process P may write. Sets L_I and L_O are reversed in process Q, which reads what P writes and writes what P reads. The system composed of P and Q running in parallel is also an LTS. The communication between P and Q is synchronous. The semantics of a composite LTS can be found in [16]. Consistency of a system is defined on the state of every process. The initial system state is always consistent, denoted by $con(s_0, \bar{s_0})$. Executing an unobservable action in P or Q does not violate the consistency of the system; see Formulas (2) and (3).

$$\frac{con(s, \bar{s}) \wedge s \xrightarrow{\tau} s'}{con(s', \bar{s})} \tag{2}$$

$$\frac{con(s, \bar{s}) \wedge \bar{s} \xrightarrow{\tau} \bar{s}'}{con(s, \bar{s}')} \tag{3}$$

If each process takes a transition with the same label in the opposite direction, consistency is preserved; see Formulas (4) and (5). For the rest of this article, we use symbols ? and ! to denote *actions* "reading" and "writing" of a given message, respectively.

$$\frac{con(s, \bar{s}) \wedge s \xrightarrow{!l} s' \wedge \bar{s} \xrightarrow{?l} \bar{s}'}{con(s', \bar{s}')} \tag{4}$$

$$\frac{con(s, \bar{s}) \wedge s \xrightarrow{?l} s' \wedge \bar{s} \xrightarrow{!l} \bar{s}'}{con(s', \bar{s}')} \tag{5}$$

Given $con(s, \bar{s})$, we say that state s *corresponds* to state $\bar{s}$, and $\bar{s}$ is a *corresponding state* of s. The state spaces of an SUT and a peer are shown in Figure 3. Label $?a$ refers to a read action of the SUT that receives message a from the peer. Labels $!b$ and $!c$ represent write actions that send messages b and c, respectively. The system starts from the consistent initial state $(s_0, \bar{s}_0)$. The SUT reads message a written by the peer, so the system state moves to $(s_1, \bar{s}_1)$. After that the SUT encounters a non-deterministic choice to write either b or c. Suppose that it writes b, the system goes to state $(s_2, \bar{s}_2)$. In a normal execution, the system terminates at this point since there is no transition left. However, if the SUT is run by a model checker, it may be backtracked to state s_1 to continue the search on an alternative path. In this case, the system is temporarily in state $(s_1, \bar{s}_2)$, which is not consistent. This state represents a situation where the peer has received message b, although the SUT has never sent it. Assuming reliable communication, the system never reaches an inconsistent state in a normal execution without external state manipulation. We therefore do not want to verify the system in such an impossible state. In this article, several approaches to avoid inconsistent states are presented.

3.4 Trace Convergence

A software model checker may report a false negative if it does not explore every possible state of the system. This is the case when a single networked process is verified by a model checker, leaving other processes executed normally by the host platform. Figure 4 shows a system of which a model checker may not find the error state, resulting in a false negative. The system starts from the initial state $(s_0, \bar{s}_0)$. The SUT has two possible actions to choose from: τ and $?a$. If the model checker selects action τ first, the system will pass state $(s_1, \bar{s}_0)$, without state change in the peer. In the next step, the SUT writes "0", and the system stays at state $(s_3, \bar{s}_4)$.

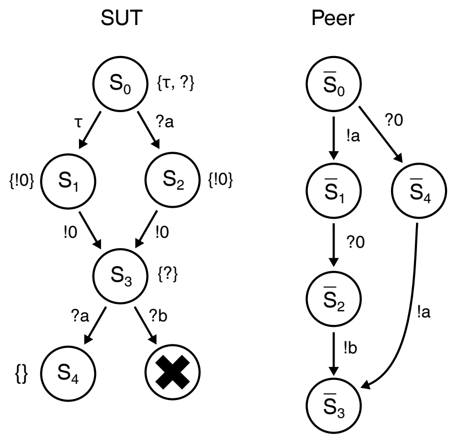


Fig. 4. A system that would cause a model checker to report a false negative. The symbols within braces show possible choices.

Finally, the SUT reads an input ($?a$) from the peer. Note that although the state space shows two possible paths, the model checker has no choice other than reading. The model checker cannot perceive the state space in way presented in the figure and cannot control external input. The system moves to state $(s_4, \bar{s}_3)$. At this point, the model checker backtracks the SUT to state s_0 since it still has an untaken choice remaining ($?a$). Assuming the presence of a mechanism to recover consistency, the system moves back to state $(s_0, \bar{s}_0)$. This time the SUT takes action $?a$, so the system moves to state $(s_2, \bar{s}_1)$ and continues to state $(s_3, \bar{s}_2)$. However, a typical model checker[1] does not continue the search, because state s_3 has no remaining path. Any error state beyond this point remains undetected. However, the SUT can reach the error state if it does not take action τ before action $?a$. The model checker therefore fails to reject the system that contains a reachable error state.

For example, a program may behave in a way described in Figure 4 if its progress is triggered by either timeout or an input. The program code is as follows.

```
m = 0
while (!timeout && m == 0) {
  if (input.isAvailable())
    m = input.read()
}
out.write("0")
m = input.read()
assert m != 'b'
```

The program waits until timeout is reached or an input is available. If it receives an input, e.g. label $?a$, before the time is over, it will continue immediately. If the timeout expires, the program, receiving no input, continues to state s_1 (label τ). The program therefore eventually moves to the state where it reads again and may fail because of a particular input.

1. On the other hand, stateless model checkers would explore both s_4 and the error state, visiting state s_3 twice.

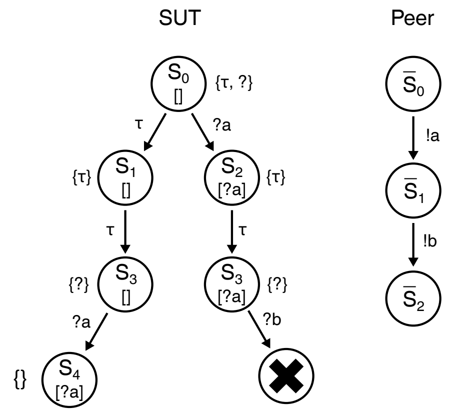


Fig. 5. The SUT state space augmented with the history of observable actions.

The false negative in this situation is caused by two event traces with different messages leading to the same SUT state. We call this phenomenon *trace convergence*. Formally, trace convergence is a situation where there exist two distinct states s, s' and two distinct event traces $\langle l_1, \ldots, l_n\rangle$, $\langle l'_1, \ldots, l'_n\rangle$ such that $s \xrightarrow{l_1,\ldots,l_n} s'$ and $s \xrightarrow{l'_1,\ldots,l'_n} s'$. The destination state of two event traces may correspond to multiple peer states, which probably emit different input traces for the SUT. In Figure 4, state s_3 is reached by two event traces with a null message (τ) and $?a$, so it corresponds to both states $\bar{s}_0$ and $\bar{s}_1$. The model checker cannot explore both possibilities since they are represented by the same state in the SUT state space.

To avoid the problem caused by trace convergence, one may augment each SUT state with message history. By doing this, the state in question is broken into two or more distinct states, depending on the number of traces converging. Figure 5 shows the SUT state space augmented with history actions. Trace convergence is eliminated since state s_3 is split into two distinct states, so the model checker can detect the error state. Note that we omit two transitions in Figure 5: the one labeled $?b$ from state $s_3[]$ and another one labeled $?a$ from state $s_3\,[?a]$. These transitions never occur in normal execution and do not need to be explored. History augmentation, however, does not guarantee verification will terminate. Since a state space may contain a loop that performs at least one observable action, the history length is unbounded, resulting in an infinite number of states. Furthermore, a false negative may still be produced if the peer is not I/O deterministic. The SUT may not encounter every possible output from the peer in this case. As a result, model checking, without knowing every possible value of peer outputs, cannot guarantee being free of false negatives and termination of verification.

3.5 Peer Restart and Replay

A mechanism to maintain the consistency of a distributed system is necessary for verifying a single process. One possible method is restarting the peer process

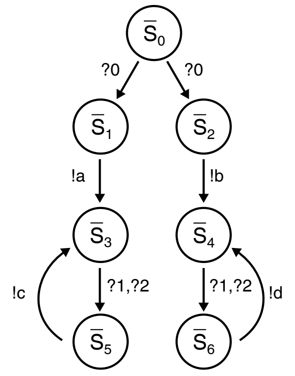


Fig. 6. The non-deterministic LTS of a peer.

when the SUT has been backtracked [13]. Since reversible execution is not possible in a normal environment, restarting may be the only way to backtrack. In the LTS, the peer will be in the initial state again. From this state, the mechanism must "replay" an interaction with the peer in a way that the peer moves to a state corresponding to the SUT state. The interaction includes both reading and writing messages. In the example shown in Figure 3, when the SUT backtracks to state s_1, the peer is restarted from state $\bar{s}_0$. To move the peer to corresponding state $\bar{s}_1$, the mechanism, on behalf of the SUT, must read message a (label $?a$) from the peer. A path to a corresponding state may contain label τ, an unobservable action. A mechanism that only controls the peer state via communication cannot force such actions to occur. We assume that the peer spontaneously performs action τ if necessary.

The problem of handling unobservable actions imposes a restriction on a peer process. If the LTS representing the peer is non-deterministic, replaying interactions does not guarantee that the peer will stay in a corresponding state. The LTS of such a peer is shown in Figure 6. Let the corresponding state be $\bar{s}_3$; it is not possible to force the peer to move to the corresponding state for certain. Although message '0' can be replayed to the peer so it can execute action $?0$, the peer has a non-deterministic choice at state $\bar{s}_0$. If it takes the path to state $\bar{s}_2$ at this point, it never reaches the destination state. Non-determinism as shown here includes randomization, thread scheduling, etc.

3.6 State Capture

The root cause of the restriction described in Section 3.5 is that the corresponding peer state of each SUT state is not preserved. The previous approach relies on the fact that the same communication trace leads to the same state. This assumption does not hold for some applications.

A more powerful method is to *capture* the corresponding state so that the peer in that state can be restored later. In practice, this can be done by checkpointing technology [17], which takes a snapshot of a controlled process and stores it in permanent storage. This snapshot contains sufficient information to create the process in a specific state. For each state (s, s') the system moves on, the snapshot of state s' is stored. When the SUT backtracks to state s, the snapshot of state s' is loaded to create the peer in that state instead of restarting the peer from the beginning. The system is then in a consistent state again. Note that this method differs from model checking the entire system in the sense that the non-determinism inside the peer process is not considered. The number of states to be explored, therefore, has the same order of magnitude, even if the state capture method is applied.

The peer LTS in Figure 6 shows state $\bar{s}_3$, which follows a non-deterministic choice. If we want to restore the peer to state $\bar{s}_3$, this implies that the system has visited consistent system state $(s, \bar{s}_3)$ before, for some SUT state s. The snapshot of state $\bar{s}_3$ is therefore available.

Compared to model checking the whole system, our approach using state capture is still modular and differs in the sense that non-determinism within the peer is not controlled. Since we only capture peer states encountered by the SUT, the number of captured states is bounded by the number of SUT states. In other words, some peer states may not be captured.

This technique does not guarantee the absence of false negatives. Only a part of the peer state space participates in verification, so the SUT may not receive all possible outputs. The peer in Figure 6 may never visit state $\bar{s}_4$ during verification, so the SUT never receives input b. If a peer output trace that is not used in the verification causes a failure in the SUT, the verification tool fails to detect the fault.

An example of such failures consists of a peer that produces a rare, yet possible, output. Some modern computer systems allow for the possibility of *leap seconds*, which sometimes add one second after midnight, resulting in time 23:59:60 [18]. Indeed, leap seconds have caused problems with the Linux Kernel as recently as June 30/July 1, 2012 [19]. Leap seconds can be considered a non-deterministic event from the point of view of the SUT. This input is rarely emitted, so the state capture approach would only consider usual inputs. The model checker would miss the cases where the SUT fails due to the unusual but possible input. Incomplete knowledge about the peer output prevents the elimination of false negatives.

3.7 Summary

Model checking of a single process of a distributed system requires state synchronization between the SUT and its peer after backtracking. Synchronization ensures that the SUT and peer processes are always in a consistent state. This section has shown two techniques for state synchronization. The first one is to restart the peer from the beginning and interact with the peer in a way that leads it to a given corresponding state. Another way

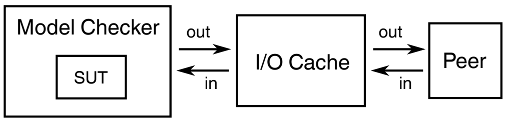


Fig. 7. The I/O cache is an intermediary between an SUT and a peer.

is to capture the peer state in a snapshot to restore the peer in a specific state later. The former is simpler to implement but is less powerful than the latter in the sense that it requires I/O determinism of the peer. Nevertheless, even with the more powerful approach of capturing states, single-process model checking may not discover all possible failures in the SUT because non-determinism in peer processes (which may result in different behavior) is not controlled.

4 Cache-based Model Checking

When model checking an SUT that communicates with a peer process, it is necessary that communication between the SUT and the peer is synchronized after backtracking. Section 3.5 has presented an approach that restarts the peer and replays the event trace to keep the peer in a consistent state. However, the overhead caused by restarting may be intolerable in practice. Implemented naively, the peer has to be restarted every time the model checker backtracks the SUT.

Cache-based model checking optimizes state synchronization, storing observed communication traces in a cache [13]. The cache reuses information from previous communication traces if possible. By doing this, the number of peer restarts is greatly reduced since the cache, given sufficient information, can interact with the SUT despite the peer in an inconsistent state. This technique significantly improves performance.

4.1 Mechanisms

In this article, the term "request" refers to a message sent from an SUT to a peer while the "response" refers to a message sent from a peer to an SUT. A cache stores a *request message* and a *response message* in pair. We call it the *I/O cache*, because it records the network input and output of each process. The I/O cache plays a role of an intermediary that intercepts every message between two processes, as shown in Figure 7. It also supplies input to both processes when necessary.

When the SUT sends a request message that is not in the cache, the I/O cache stores this message in a data structure and forwards it to the peer. After a specific amount of time, the I/O cache polls the peer for a response. If there is no response, the I/O cache assumes that the current request is *incomplete*. The peer would need more requests to produce a response. If a response comes back, the cache will associate the pending incomplete requests, if any, and the latest request with the response. The association of requests and a response

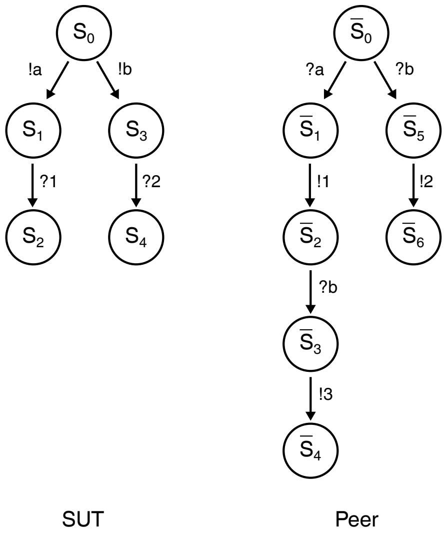


Fig. 8. LTS representing an SUT and a peer that must be restarted during verification

in the I/O cache is called a *cache entry*. The observed (cached) response is taken as the earliest response that corresponds to a given request [12]. Other schedules with later responses are automatically generated by Java PathFinder; also see Section 5.

A list of cache entries represents a partial communication trace between two processes. An SUT with deterministic output still produces the same trace after backtracking. Rather than restarting the peer, the I/O cache uses the information from previous traces to interact with the SUT. The method avoids restarting the peer and improves the performance of verification. Note that the I/O cache cannot decide if the peer is going to produce a response for the current request. The peer is assumed to send a response within a certain amount of time or not at all. If this assumption fails, the delayed response message may mix up with the next response message. The resulting cache entry corresponds to a real system execution but does not take into account the possibility of a faster peer response; this may result in a false negative.

While responses for a previously seen request can be taken from the cache, the peer has to be restarted when the SUT sends a different message trace after backtracking. Suppose that we have an SUT and a peer modeled by the LTS in Figure 8. When the SUT sends request 'a', the I/O cache polls for response '1' and creates cache entry $\langle a, 1\rangle$. After backtracking, the SUT sends request 'b'. The I/O cache restarts the peer and sends message 'b' to the peer, resulting in cache entry $\langle b, 2\rangle$. Note that if the I/O cache did not restart the peer before sending a new message trace, it would get response '3' for request 'b', which was incorrect.

The I/O cache also keeps track of the state of the SUT so that it can supply correct input. Each state of the SUT corresponds to positions in the input and output traces, so the I/O cache has a pair of *state pointers* that point

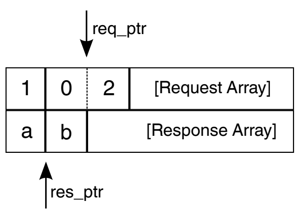


Fig. 9. The data structure of the linear cache.

to a communication trace. A pair of state pointers is composed of a *request pointer* and a *response pointer*. The request and response pointers refer to the last message that the SUT has sent and received, respectively. These pointers are moved according to the state of the SUT. When the SUT is backtracked by a model checker, the state pointers are reverted to the corresponding positions.

The I/O cache can be implemented as two variants: *linear cache* and *branching cache*.

4.2 Linear Cache

A linear cache [12] models communication data using a pair of arrays. Two arrays, called a *request array* and a *response array*, store request messages and response messages, respectively. Figure 9 shows a representation of the linear cache for a system where request 1 elicits response a, and a subsequent request $\langle 0, 2\rangle$ elicits b.

A cache entry is a pair of contiguous elements in the request array and an element in the response array. Figure 9 displays two cache entries. The first one contains pair $(\langle 1\rangle, a)$, and another one contains pair $(\langle 0, 2\rangle, b)$. The state pointers move along both arrays. A pair of array indices represents state pointers. The state pointers in Figure 9 are denoted by $(2, 1)$. This is a state where the request is incomplete, so the SUT cannot read response b until request 2 is sent. Every SUT state associates with a pair of state pointers [12].

The linear cache requires that the SUT produce deterministic output, as defined in Section 3.1. If this property does not hold, the SUT may emit multiple output traces under different schedules. The linear cache cannot store multiple communication traces because of its linear data structure. Some applications produce non-deterministic output such as a dynamic web server that includes a visitor count in the output. Such applications cannot be verified by the linear cache. For cases the SUT only produces one output trace, the linear cache can interact with the SUT without restarting the peer at all, after it has captured a complete communication trace.

4.3 Branching Cache

The restriction of the linear cache can be relaxed by changing the linear data structure to a tree structure [13]. Branches in the tree store multiple communication traces. Figure 10 shows the tree structure used in the branching cache. Every non-root node in the tree is

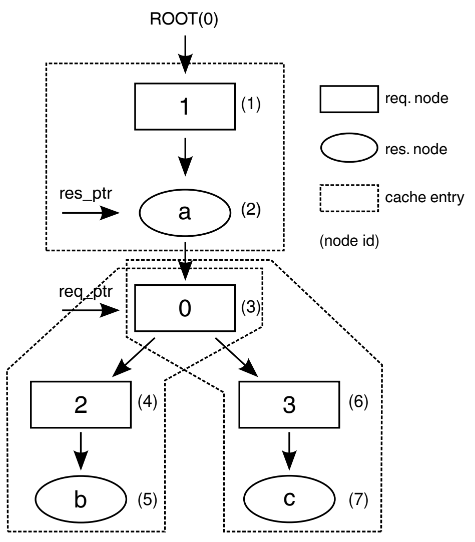


Fig. 10. The data structure of the branching cache.

classified into either a *request node* or a *response node*. A request node can have either one or more request child nodes, or one response node, but not both. A response node can only have one or more request child nodes. We wrap each response message, which may contain multiple characters, in a single response node. A group of contiguous request nodes represents a complete request message. The following response node, if any, represents the associated response message. A *cache entry*, therefore, contains the group of contiguous request nodes and a response node.

The branching cache must restart a peer when it receives a new output trace from an SUT. Since the branching cache has not yet seen the new output trace, it does not know the corresponding response. The running peer, which is not in a corresponding state, is terminated. A new peer instance is started and supplied with the new output trace. The branching cache polls the peer for a response and stores this information in a new branch in the same way. Figure 10 shows two output traces of an SUT: $\langle 1, 0, 2\rangle$ and $\langle 1, 0, 3\rangle$. The peer is restarted once to handle the second output trace. In general, if the SUT produces n distinct output traces, the number of peer restarts is $n - 1$.

4.4 Limitations

The branching cache is more powerful than the linear cache in the sense that it supports SUT with non-deterministic output. However, it cannot guarantee that the new instance of the peer will behave in the same way as the original peer would due to the limitation of the peer-restart approach described in Section 3.5. The capabilities of both types of cache shown in this section are displayed in Figure 11. The linear cache supports applications with both an SUT and a peer being I/O deterministic. The branching cache extends the coverage to I/O non-deterministic SUT, because it stores multiple communication traces.

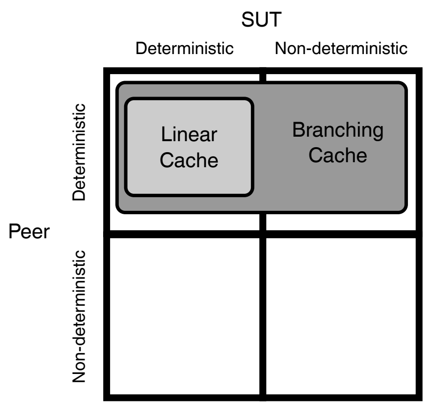


Fig. 11. Application classes in terms of I/O determinism supported by the linear cache and the branching cache.

The I/O cache assumes that a peer sends a response, if any, within a certain amount of time. Without prior knowledge of the protocol and peer behavior, this provides the best approximation of the peer response. If a peer is slow, one would need a raise of the waiting time to prevent false matching. Note that incomplete responses still represent valid communication traces, as a network can be slow; however, they respond a subset of all possible communication traces, ignoring the cases with low latency.

5 Implementation using Java PathFinder

Java PathFinder (JPF) is a model checker that executes Java bytecode [6], [20]. It is equipped with a custom *Java virtual machine (JVM)* [21] for running Java bytecode. This custom JVM is written in Java and operates on top of a host JVM. It is capable of rolling back the state of a running Java program so that JPF can re-execute the program along other execution paths. Multiple execution paths are generated from several sources of non-determinism in the program such as thread scheduling and choice generators modeling a possible range of data values. If JPF finds a state that violates a specified property, it will report an error trace showing execution from the initial state to the error state. The user can refer to the error trace to track down and fix the program fault. By default, JPF uses *depth-first search* to explore the program state space.

We implemented cache-based model checking presented in Section 4 as the software model checker extension called *net-iocache* [8] on top of JPF. This extension can be downloaded from [22]. JPF explores the program state space while the extension works as an I/O interface to external peer processes for an SUT. Every message transferred between the SUT and peers is monitored by the I/O cache. The I/O cache emulates peer behavior by supplying the SUT with cached messages whenever possible.

A Java program creates a `ServerSocket` instance to accept a connection at a specified port from another process. On the other hand, another Java program can make a connection by creating a `Socket` instance with a destination address and port number. When a connection has been established, a process can send a message to another process by writing the message into the output stream associated with the connection. Similarly, the process reads a message from the input stream. The read operation is blocking: thread execution is blocked until an incoming message is available or the connection is closed. The write operation is non-blocking; thread execution continues without waiting for another process to receive the data, as the message is stored in a buffer on the destination host. However, the I/O cache blocks JPF internally until a response message arrives or the timeout expires. This small timeout is not visible to the SUT as JPF itself is blocked, preventing any other SUT threads from executing. This communication mechanism can be mapped to our formalization presented in Section 3.3, where the communication between two LTS is synchronous.

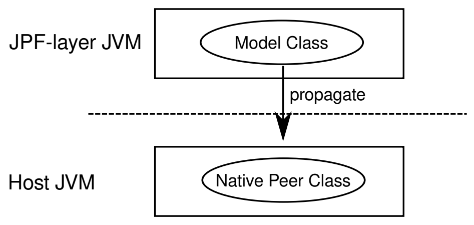


Fig. 12. Method call propagation between virtual machines.

5.1 Extension Mechanisms

JPF has been designed to encourage extensions. An extension may add new functionality to the model checker. JPF provides three extension mechanisms: *choice generators*, *listeners*, and *Model Java Interface (MJI)* [23]. Listeners and MJI are briefly described as follows.

Listeners allow extensions to observe events occurring inside JPF during execution. A listener tracks the state space search by listening to events that are emitted when JPF starts a search, forwards to a new state, backtracks, and finishes the search.

The Model Java Interface (MJI) allows users to run a certain piece of code on the host JVM, rather than the custom JVM. This mechanism becomes useful in several situations. For example, a piece of code that is not subject to verification does not require the JPF state-tracking function. Running the code on the host JVM causes JPF to execute it atomically, omitting unnecessary decisions in the state space. This can be done by propagating the call to the host JVM level, as shown in Figure 12. MJI is also essential in implementing classes that are not subject to backtracking, such as cache contents. When backtracking, JPF only reverts the program state tracked by the JPF virtual machine. The program data on the host JVM remains intact.

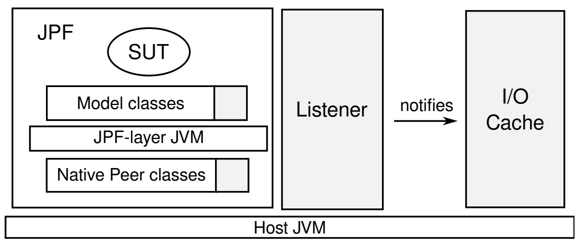


Fig. 13. Design of `net-iocache`.

The Java memory model allows violation of *sequential consistency* [24]. However, JPF, by default, only supports interleaving semantics. Faults caused by instruction reordering (due to optimization) may result in false negatives. Sequential consistency can be guaranteed if a system is free of data races. *Java RaceFinder (JRF)* [25] is a JPF extension that detects data races in multithreaded programs. One may use this tool to ascertain data race freeness before analyzing a system further.

5.2 Project net-iocache

We develop the JPF extension `net-iocache` [22], which supports verification of distributed systems, based on cache-based model checking. The branching I/O cache is used as the data structure to store communication traces. The design is shown in Figure 13. Every component runs on a host JVM except for the SUT. The SUT communicates over the network using the standard Java API; model classes intercept the library calls and relay communication to the I/O cache.

The I/O cache keeps track of state changes in JPF, in order to properly respond to the SUT while the program state space is being explored. This is implemented by registering two listener to JPF: one that notifies the I/O cache when a state transition takes place and one that collects verification information as JPF makes progress. State changes are captured by three events: `stateAdvanced`, `stateBacktracked`, and `stateRestored`.

JPF itself (the core module) does not support the Java network library, so package `java.net` cannot be executed. We provide a replacement for that library and its internal (native) code using the Model Java Interface. For each missing class, we create a *model class* and a *native peer class* [23]. The model class stores information associated with the SUT state while the native peer class performs actual I/O tasks such as writing and reading. The model class can selectively delegate calls to certain methods to the corresponding native peer class, as shown in Figure 12.

Two model classes for stream objects, *CacheLayer-Input/OutputStream*, allow the I/O cache to control input/output messages of SUT. Figure 14 shows an overview of the communication between the components in a verification system. These stream objects interact with the SUT in the same way that the actual peer would. The model class for class `Socket` returns these custom stream objects when requested by the SUT. Every message written via `CacheLayerOutputStream` is redirected to the I/O cache instead of the peer. Similarly, the SUT reads messages via `CacheLayerInputStream`, which the I/O cache fills input into, rather than the real peer. The I/O cache holds standard sockets including standard input/output streams connected to the peer. Note that these model classes are ready to use; testers do not have to write any models before verification. This is not the case for NetStub, which requires testers to write a model for a specific peer process.

6 Application of Checkpointing Tools

The I/O cache described so far supports I/O deterministic SUTs (using the linear cache) or I/O non-deterministic SUTs (using the branching cache) communicating with deterministic peers. A peer is restarted if SUT behavior diverges (in the branching cache).

However, the peer restart method discussed in Section 3.5 does not support peers that produce non-deterministic output. A new instance of the peer may not be in the same state that a previous instance was in, even after replaying the same communication trace to the same program, due to non-determinism. Taking a snapshot of the corresponding peer state for each SUT state overcomes the problem of peer non-determinism. To implement this idea, checkpointing technology comes into use, although it cannot handle non-determinism in thread scheduling, as mentioned in Section 3.6. This section introduces how to apply a checkpointing tool to software model checking.

6.1 Process Checkpointing

Process checkpointing [14] is a technique to create a snapshot of one or more processes. The snapshot is usually called a *checkpoint*. A checkpoint is typically stored as a file and can be loaded later to restore the subject in a certain state. The restored processes continue running from where they were suspended as if they had not stopped running.

Most *virtualization tools* [26], [27], [28] such as *Kernel-based Virtual Machine* (*KVM*) provide basic checkpointing functions: save and restore. However, virtualization consumes a large amount of system resources since the subject of checkpointing is an entire operating system. Creating a checkpoint by these tools may take a considerable amount of time even for a small peer process. On the other hand, virtualization tools are powerful execution environments since they usually preserve the state of the operating system. This functionality may be required by some peer programs.

Checkpointing at process level is more scalable, because the overhead of handling unrelated OS processes is eliminated. *MultiThreaded CheckPointing* (*MTCP*) [17], a process-level checkpointing tool, can be used to

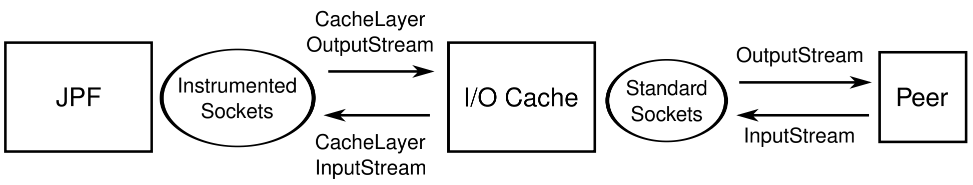


Fig. 14. Communication between components in a verification system.

TABLE 1
Comparison of KVM, MTCP, and DMTCP.

Properties	**Checkpointing Tools**		
	KVM	**MTCP**	**DMTCP**
Virtualization level	OS	single-process	group of processes
Disk images	reversible	irreversible	irreversible
Process info	preserved	not preserved	preserved
Port numbers	preserved	not preserved	not preserved

create snapshots of a peer if the peer is a single-process program. *Distributed MultiThreaded CheckPointing* (*DMTCP*) [29] is an extended version of MTCP that manages a group of processes connected by network connections or parent-child relations. This tool can be used to create a checkpoint of multiple peer processes executing simultaneously. The `net-iocache` project currently employs DMTCP as the checkpointing environment to support peers emitting non-deterministic output. The tool is briefly explained in Section 6.5.

Table 1 summarizes important properties of the tools. Each of them controls target systems at different levels. Only KVM can preserve changes on a file system since the entire operating system is checkpointed. MTCP does not include information about process (`/proc`) in a checkpoint. As a result, peers may find that their process identifiers change from time to time. Finally, MTCP and DMTCP only bind a listening socket to the same port number after restoring a checkpoint. Local port numbers used by a process may change.

Each tool is different in the capability of supporting verification, as shown in Table 1. One should use a tool based on their systems to be verified. In our work, we chose DMTCP in our implementation since it is sufficiently powerful for most systems in general.

6.2 Integration with Software Verification

Checkpointing introduces another method to synchronize a peer with an SUT. Model checkers save SUT states in order to backtrack it to any previously visited point in the state space. A checkpointing tool can do the same with peer processes. When the SUT backtracks to a previous state, the checkpointing tool restores the peer process by the checkpoint of the corresponding state, instead of restarting the peer from the beginning. In the extreme case, we may create a peer checkpoint for each SUT state. In practice, the peer does not have to be checkpointed as often as the SUT. Some checkpoints can be omitted under certain conditions. Optimizations taking advantage of this fact are discussed in Section 6.4.

Process checkpointing tools also allow monitoring of the system that is executed. Monitoring can capture non-determinism in peers. Such non-determinism can be divided into two types: thread scheduling and external input. Thread scheduling is controlled by the operating system. We only observe one peer execution for each input and do not control the schedule of peer processes. Hence, we assume peer output of each communication channel is independent of thread scheduling. This assumption is reasonable since programmers expect this property in most cases. Other tools can verify this property; see Section 8.

A peer process may receive input both externally and internally from several sources during execution. The input from an SUT is explicit. If the peer produces the same outgoing trace for each incoming trace from the SUT, we call that its output I/O deterministic, as defined in Section 3.1. Peers with non-deterministic output also take implicit input into account during computation. The sources of such input are the facilities provided by the operating system on which the peer is running such as system time, the file system, peripheral devices, etc. Some special files serve as random number generators such as `/dev/random` and `/dev/urandom` in Unix-like operating systems. Non-deterministic input from such files has an effect on the peer output. Implicit input can be detected by inspecting peer's calls to certain functions that take external input. This inspection lets us know when non-determinism occurs in a peer.

Although the checkpointing tool provides a simple framework to control the peer state, it comes at a price. Saving and restoring a peer state are expensive operations since they involve I/O operations. Furthermore, if the checkpointing tool saves the peer state at every transition, the overhead from checkpointing will grow with the size of the state space. The scalability of such verification systems is difficult to maintain.

6.3 Hybrid Approach

In a *hybrid approach,* the I/O cache can be combined with the checkpointing tool to reduce the overhead of checkpointing. The I/O cache stores request and response messages while the peer process executes under the control of the checkpointing tool. A peer checkpoint is created every time the SUT reaches a new state. However, when the SUT is backtracked, we do not

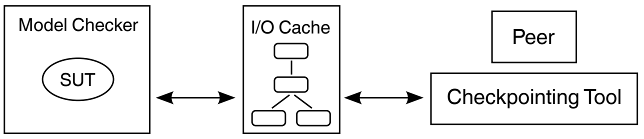


Fig. 15. The configuration of the hybrid approach.

necessarily need to recover the corresponding peer state from a checkpoint. The I/O cache can supply data to the SUT for any previously observed requests. In other words, if the response message for the current output trace of the SUT is in the I/O cache, the I/O cache uses this message to interact with the SUT. No checkpoint is loaded in such cases. Checkpoint recovery only takes place when the SUT produces a hitherto unseen output trace. In this case, an appropriate checkpoint must be restored, as in the pure checkpointing approach.

The hybrid approach greatly reduces the overhead since with caching, the number of checkpoint restarts depends on the number of distinct output traces of the SUT, rather than the size of the state space. Figure 15 shows the configuration of the hybrid approach. The I/O cache is between the model checker and the checkpointing tool. It is also responsible for controlling the checkpointing tool to save/restore the peer process.

6.4 Checkpointing Strategies

The number of checkpoints can be reduced since not all of them are essential [8]. The concept of *logical checkpoints* is introduced to perform this optimization. A logical checkpoint corresponds to the state of a peer program at a given time. For each SUT state, there is a logical peer checkpoint. Therefore, we can build a state space of logical checkpoints similar to the state space of the SUT. However, logical checkpoints are not always backed by an actual (physical) data structure. In contrast to this, *physical checkpoints* are real and contain sufficient information for restoring a peer in a specific state. The two types of checkpoints form a structure, shown in Figure 16, similar to the SUT state space. A *checkpointing strategy* defines how to maintain the balance of the checkpoint creation overhead with the possibility of restoring a previous state directly. It decides whether to create a physical checkpoint over the corresponding logical checkpoint. For example, we could reasonably assume that the peer does not change its state if the SUT does not interact with it during a state transition. Thus, peer checkpointing could be skipped after such a transition. Figure 16 shows the checkpoint space of a peer where some checkpoints are omitted. Since the peer does not change state, logical checkpoints are equivalent to the latest physical checkpoint of their ancestor states. According to Figure 16, checkpoint C_0 is equivalent to other three checkpoints C_1, C_4, and C_5. By this assumption, physical checkpoint C_i is created only if the SUT state transition to S_i involves communication with the peer. This strategy is called *io-only*.

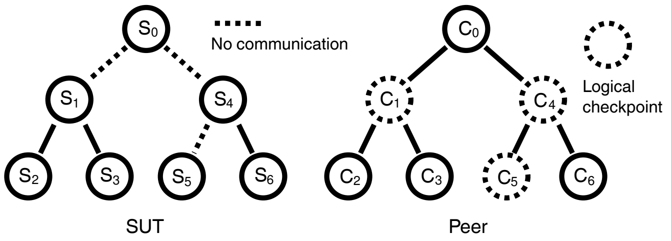


Fig. 16. The state space of a SUT and the checkpoint space of a peer.

Some checkpoints are essential in maintaining consistency of cached data. The checkpointing tool must sufficiently create checkpoints of the peer such that the SUT observes exactly one input trace for each output trace. The I/O cache requires that peers produce deterministic output. Consider a case where a peer produces multiple outcomes according to the transition system shown in Figure 6 (Section 3). The peer may produce a different output ('a' or 'b') in each run, although it receives the same request '0'. Suppose that the peer emits 'a' for the first request and 'c' for the second request. As a result, the I/O cache contains two entries at this point: $(\langle 0\rangle, a)$ and $(\langle 1\rangle, c)$. The SUT is backtracked and produces a diverging trace ('02'). If a peer checkpoint at state $\bar{s}_1$ or $\bar{s}_3$ does not exist, the peer will be restarted from the beginning. During a second execution, the peer then possibly returns message 'b' instead, which does not match the first cache entry, as shown in Figure 17. The I/O cache may either abort, or use the existing response 'a' after this point. If execution continues, the peer instance may return 'd' for the second request. The new cache entry to be added is $(\langle 2\rangle, d)$. Clearly, the I/O cache now contains an impossible input trace since the peer never returns 'ad', according to its LTS shown in Figure 6. Note that the same problem occurs, even if the I/O cache uses response 'b'. The fact that the I/O cache cannot contain complete information about peer responses prevents us from perfectly emulating the peer behavior. Such a non-deterministic peer may cause a false positive in model checking by supplying an input trace that never happens in the real system.

This situation can be avoided by creating a checkpoint after each execution of non-deterministic instructions. In Figure 6, a non-deterministic instruction is executed during the transition from $\bar{s}_0$ to either $\bar{s}_1$ or $\bar{s}_2$. The peer therefore is checkpointed at state $\bar{s}_1$ or $\bar{s}_2$, depending on which is selected in the first run. The outcome of the non-deterministic decision is recorded in the checkpoint, so the non-deterministic decision will not be made again; therefore, the peer output sent to the SUT is consistent. The checkpoints created by this condition form the smallest set of checkpoints required to support non-deterministic peers. The checkpointing strategy that only creates a checkpoint after a peer non-deterministic instruction is called *nd-only*.

Peer restart may also be considered a checkpointing

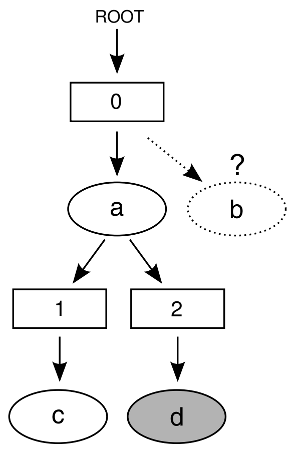


Fig. 17. Inconsistency in the cache data structure.

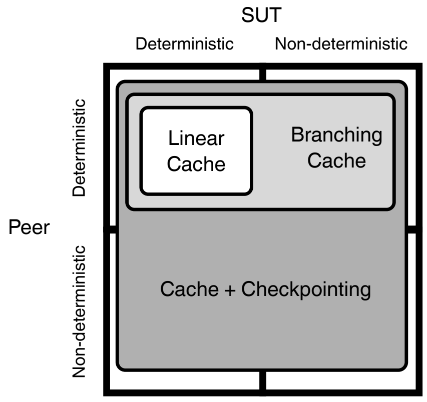


Fig. 18. Application classes supported by each approach.

strategy where no checkpoint is created. This strategy does not support non-deterministic peers, as discussed in Section 3.5.

With checkpointing support, the cache-based model checking can verify every class of applications shown in Figure 18. Although some restrictions apply (see Sections 3.4 and 3.6), this technique is useful for finding faults in the implementation of distributed systems.

6.5 Underlying Mechanisms

To be used for cache-based model checking, a checkpointing tool must provide some essential features. One of them is to recover communication channels among peer processes when a checkpoint has been loaded. The checkpointing tool may establish a new connection behind the scene and assign a new socket descriptor to the process that had owned the connection. This operation is done transparently, so the peer process does not have to do anything special before using the connection after state recovery.

Another feature that we have mentioned is a capability of detecting non-deterministic operations inside the peer process. Although most checkpointing tools do not provide this feature, we can write a set of wrappers of functions that cause non-deterministic results. These wrappers, when invoked, must notify the verification system so that it creates the checkpoint of the current peer state. In this paper, we use the checkpointing tool called DMTCP in our implementation.

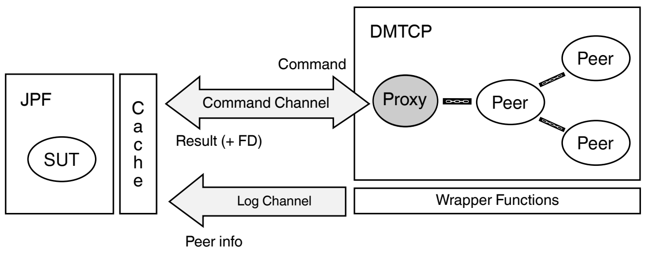


Fig. 19. The proxy process represents the SUT inside the DMTCP environment. The I/O cache has two communication channels connected to DMTCP.

DMTCP is a checkpointing tool for a group of processes. All processes are executed in a special environment where a number of standard functions are wrapped in order to gain information for creating system checkpoints. Each process in a group is called a *node*. If a node creates a new process by a `fork`-family function, the child process will also become a node in the same group. DMTCP saves the entire state of the process group including connections among the internal processes when receiving the checkpoint command. Similarly, it restarts all processes in a group from a given checkpoint when receiving the restart command.

The SUT state is controlled by JPF while the peer state is controlled by DMTCP. Since the SUT is not a process in the group, the connection between the SUT and peers is not subject to checkpointing. As a result, the lifetime of the connection is over when the I/O cache kills the group of peers before loading a new one from a checkpoint. When restarting, the I/O cache must restore this connection so that the SUT and peer can communicate with each other again.

A *proxy process* makes the connection recovery problem simpler by letting DMTCP manage all connections between the SUT and peers [8]. It represents the SUT in the DMTCP environment and works similar to other nodes in the group as shown in Figure 19. When the SUT opens or closes a connection, the I/O cache sends the corresponding command to the proxy process. The proxy performs the requested operation and sends the result back to the I/O cache. Some operations such as `accept` return a file descriptor that represents a socket. The I/O cache uses the file descriptor it receives to directly communicate with the peer.

DMTCP provides a set of wrapper functions that collects necessary information for checkpointing before calling the real version of the functions. The wrapped functions include both standard C libraries and system calls. In a similar way, we add a wrapper for each function that may cause non-deterministic behavior. When one of these wrappers detects non-determinism, it sends a notification to the I/O cache via the *log channel* shown in Figure 19. The I/O cache therefore knows that the current peer state is essential and must be captured in a checkpoint. This mechanism is used in every checkpointing strategy described in Section 6.4.

7 EXPERIMENTS

The previous sections have shown several approaches to the verification of a single process in a distributed application using the I/O cache: caching with a linear or a branching data structure, and a hybrid approach with a checkpointing tool. Each approach supports a different class of applications, as shown in Figure 18. The I/O cache with checkpointing, although the most powerful, comes with the expense of overhead in the checkpointing operations. The overhead is reduced by applying a checkpointing strategy, which selectively takes snapshots of peer processes. The performance of each approach is measured and presented in this section. Experiments are set up to compare the time spent in the verification of several distributed applications. The experiments on the checkpointing cache are further divided based on the checkpointing strategy to show how the performance is affected.

7.1 Experimental Environment

Every experiment is run on an 8-core Mac Pro workstation with 24 GB of physical memory, running Ubuntu 10.10. Although distributed systems are subjects for the experiment, we do not have to run them on multiple machines. All processes are run in a single machine in order to prevent network latency from affecting the experimental results. In practice, a tester would prefer to do the same thing if possible. The time limit for each case is set to one hour. JPF (`jpf-core` module) revision 679 and DMTCP 1.2.4 are used in the experiment.[2] JPF is configured to detect three types of faults: assertion violations, deadlocks, and uncaught exceptions. Our JPF extension net-iocache works in two modes: normal mode and checkpointing mode. The normal mode natively runs a peer process on a host OS and uses the I/O cache to emulate the peer behavior whenever possible. The checkpointing mode starts a peer in DMTCP, together with a proxy process (see Section 6.5). The I/O cache is still used to capture communication traces to reduce overhead in this mode, as described in Section 6.4. Currently, DMTCP is not fully compatible with the Java Virtual Machine, so we use the peers written in C for the experiment run in the checkpointing mode. Table 2 shows the list of applications used in the experiments, including processes verified by JPF (SUT) and environmental processes (peers). Some processes have non-deterministic version for the experiments that demonstrate support for non-determinism in the output.

2. A snapshot of the sources can be found at http://staff.aist.go.jp/c.artho/tse/.

7.2 Results

The experimental result of applications composed of deterministic SUTs and deterministic peers is shown in Table 3.[3] Two data structures, linear and branching cache, are compared in this experiment. The number of states explored by JPF increases as the system becomes more complex due to the number of threads increasing. The SUTs used in this experiment do not produce any non-deterministic outputs, so the branching cache is not strictly necessary. We show that the minor difference in time spent can only be seen in large test cases. This shows that the branching cache adds a very small computational cost to the verification system, while being able to handle non-deterministic SUTs.

The second set of our experiments is performed on applications with non-deterministic SUTs. The branching cache and the hybrid approach with DMTCP were used to verify these applications. We further arrange the experiment of the hybrid approach into three configurations based on the checkpointing strategy. Table 4 shows the experimental results. Although the checkpointing approach is not necessary for deterministic peers, this experiment shows that its performance is usually comparable to the cache-based approach, given a good checkpointing strategy.

Table 5 shows experimental results on the applications with non-determinism in both the SUT and the peer. Only the hybrid checkpointing approach is applicable in this experiment. Three checkpointing strategies are compared. The *all* strategy creates a checkpoint for every SUT state and is used as a baseline for our study. According to the result, the *io-only* and *nd-only* strategies greatly reduce the number of checkpoints generated during verification.

7.3 Discussion

The branching cache shows performance comparable to the linear cache despite additional complexity in the data structure. Therefore, it is useful for all cases where all peers are I/O deterministic.

Systems where peers are I/O non-deterministic require the use of the hybrid checkpointing approach. Given an efficient checkpointing strategy, the overhead introduced by checkpointing is minor. Tables 4 and 5 show that checkpoint-based model checking with JPF and DMTCP can verify more than two million states in less than 30 minutes.

The performance of the checkpointing approach with the *nd-only* strategy is not much different from the branching cache approach since it only creates a checkpoint if necessary. It also provides support for non-deterministic peers, making it more powerful. The *io-only* strategy is usually slower than *nd-only*, because it creates

3. The version of caches used in these experiments is newer than the version used in our previous publications [8], [13]. However, all versions share a common code base.

TABLE 2
The list of applications in the experiments and descriptions.

Application		Description
SUT	**Peer**	
Alphabet client	Alphabet server	The alphabet client contains *producer* and *consumer* threads. Each producer thread creates a connection and sends a specified number of messages to the alphabet server; each message consists of a number n followed by a newline character. The client consumer threads read incoming messages. The non-deterministic version of the client randomly selects one of two patterns of request messages to send to the server.
Alphabet server	Alphabet client	The server creates a thread for serving each incoming connection, which returns the nth letter of the Latin alphabet followed by a newline character. The non-deterministic version of the server randomly sends either an uppercase or lowercase nth letter for each message n.
HTTP client	HTTP server	The HTTP client requests a page from a server via HTTP. It generates a number of worker threads to request multiple documents in parallel. Each worker thread creates a connection to the HTTP server. The non-deterministic version of the client randomly requests one out of two given pages.
HTTP server	HTTP client	The HTTP server creates a thread for serving each incoming connection. The server is written in Java and processes basic HTTP requests. The non-deterministic version of the server attaches a visitor counter to the output. The client, therefore, may receive a different content depending on the counter, although requesting the same page.
HTTP client	thttpd [30]	*thttpd* is a lightweight HTTP server written in C. This application is used in the experiment with DMTCP.
Time client	Time server	The time client creates a specified number of threads. Each thread creates a connection to the time server. A client thread fails if it encounters a leap second. This failure may not be detected by the branching I/O cache, causing a false negative (see Section 3.6). The time server is a single-threaded program. It returns the current time when accepting a connection, without an explicit request message from the client. The non-deterministic version of the time server may return time with a leap second.

TABLE 3
The experimental results of deterministic SUT and deterministic peers.

SUT	Peer	#conn	#msg	time (mm:ss)		#states
				linear	branching	
Alphabet client	Alphabet server	2	2	0:01	0:01	755
		2	3	0:01	0:01	930
		2	4	0:01	0:01	1,113
		3	2	0:08	0:08	13,015
		3	3	0:09	0:09	16,522
		3	4	0:10	0:10	20,037
		4	2	2:14	2:14	250,652
		4	3	2:32	2:32	329,299
		4	4	2:48	2:50	407,954
		5	2	47:42	47:51	4,849,344
Alphabet server	Alphabet client	2	2	0:01	0:01	53
		2	3	0:01	0:01	61
		2	4	0:01	0:01	69
		3	2	0:01	0:01	337
		3	3	0:01	0:01	481
		3	4	0:01	0:01	655
		4	2	0:03	0:03	2,512
		4	3	0:05	0:05	4,483
		4	4	0:06	0:06	7,340
		5	2	0:14	0:15	17,714
		5	3	0:30	0:30	39,114
		5	4	0:55	0:56	76,124
HTTP client	HTTP server	2	1	0:03	0:04	892
		3	1	1:26	1:36	10,948
		4	1	> 1h	> 1h	–
HTTP server	HTTP client	2	1	0:01	0:01	241
		3	1	0:04	0:04	2,448
		4	1	0:41	0:44	28,666
Time client	Time server	2	1	0:01	0:01	139
		3	1	0:03	0:03	2,145
		4	1	1:42	2:17	95,898
		5	1	> 1h	> 1h	–

TABLE 4
The experimental results of non-deterministic SUT and deterministic peers.

SUT	Peer	#conn	#msg	time (mm:ss) [#checkpoints]				#states
				branching	checkpointing			
					all	io-only	nd-only	
ND alphabet client	Alphabet server	2	2	0:02	6:47 [1,018]	0:08 [6]	0:06 [1]	2,326
			3	0:06	46:31 [7,002]	0:16 [11]	0:12 [1]	9,945
			4	0:18	> 1h	0:38 [21]	0:29 [1]	43,960
		3	2	0:37	> 1h	0:47 [10]	0:42 [1]	73,574
			3	4:26	–	4:51 [19]	4:44 [1]	623,437
			4	36:25	–	38:11 [37]	38:10 [1]	5,507,260
		4	2	21:30	–	22:13 [14]	22:07 [1]	2,425,706
			3	> 1h	–	–	–	–
ND alphabet server	Alphabet client	2	2	0:01	0:53 [114]	0:09 [6]	0:07 [1]	120
			3	0:02	1:16 [156]	0:13 [11]	0:09 [1]	160
			4	0:02	1:31 [196]	0:17 [15]	0:12 [1]	200
			5	0:04	1:53 [236]	0:24 [19]	0:14 [1]	240
		3	2	0:03	10:14 [1,460]	0:14 [11]	0:12 [1]	1,464
			3	0:04	12:02 [1,716]	0:20 [17]	0:17 [1]	1,720
			4	0:06	27:14 [3,908]	0:27 [23]	0:22 [1]	3,912
			5	0:11	38:30 [5,552]	0:33 [29]	0:28 [1]	5,556
		4	2	0:11	> 1h	0:22 [15]	0:23 [1]	15,692
			3	0:19	–	0:41 [23]	0:39 [1]	34,036
			4	0:34	–	1:01 [31]	0:59 [1]	62,732
			5	0:53	–	1:28 [39]	1:28 [1]	104,108
		5	2	1:27	–	1:46 [19]	1:46 [1]	157,355
			3	3:50	–	4:26 [29]	4:20 [1]	427,127
			4	8:52	–	9:20 [39]	9:19 [1]	949,523
			5	16:38	–	17:49 [49]	17.44 [1]	1,849,343
HTTP client	HTTP server	2	1	0:06	12:27 [1,867]	0:11 [5]	0:10 [3]	1,867
		3	1	8:44	> 1h	8:54 [7]	8:52 [4]	78,248
HTTP server	HTTP client	2	1	0:01	0:11 [9]	0:08 [3]	0:06 [1]	267
		3	1	0:07	0:36 [31]	0:19 [5]	0:18 [1]	3,366
		4	1	1:21	2:55 [71]	2:13 [7]	2:13 [1]	48,518
		5	1	24:32	37:29 [137]	36:06 [9]	36:04 [1]	780,557

TABLE 5
The experimental results of applications whose peers are non-deterministic.

SUT	Peer	#conn	#msg	time (mm:ss) [#checkpoints]			#states
				checkpointing			
				all	io-only	nd-only	
ND alphabet server	ND alphabet client	2	2	0:51 [114]	0:10 [9]	0:10 [8]	120
			3	1:10 [156]	0:15 [14]	0:14 [13]	160
			4	1:28 [196]	0:18 [18]	0:18 [17]	200
			5	1:47 [236]	0:22 [22]	0:21 [21]	240
		3	2	9:52 [1,460]	0:16 [15]	0:15 [13]	1,464
			3	17:11 [2,544]	0:21 [21]	0:21 [19]	2,548
			4	26:15 [3,908]	0:29 [27]	0:28 [25]	3,912
			5	37:21 [5,552]	0:35 [33]	0:34 [31]	5,556
		4	2	> 1h	0:27 [21]	0:27 [18]	15,692
			3	–	0:43 [29]	0:42 [26]	34,036
			4	–	1:03 [37]	1:02 [34]	62,732
			5	–	1:34 [45]	1:32 [42]	104,108
		5	2	–	1:48 [27]	1:48 [22]	157,355
			3	–	4:31 [37]	4:22 [33]	427,127
			4	–	9:24 [47]	9:24 [43]	949,523
			5	–	17:57 [57]	17:52 [53]	1,849,343
ND alphabet client	ND alphabet server	2	2	6:47 [1,018]	0:08 [6]	0:08 [4]	2,326
			3	46:39 [7,002]	0:16 [12]	0:14 [7]	9,945
			4	> 1h	0:40 [24]	0:34 [13]	43,960
		3	2	> 1h	0:46 [10]	0:45 [6]	73,574
			3	–	4:55 [20]	4:49 [11]	623,437
			4	–	38:20 [40]	38:07 [21]	5,507,260
		4	2	–	22:26 [14]	22:01 [8]	2,425,706

more checkpoints. However, it would be more advantageous if the peer took a long time between I/O operations. In this case the *io-only* strategy would prevent re-execution of expensive computations between I/O operations. The *all* strategy excessively creates checkpoints and gives poor performance, so the strategy itself is not practically useful but included as a baseline for reference.

7.4 Case Studies

The cache-based model checking approach was used in various case studies. Notable applications are (1) *jget,* a parallelizable download client, (2) a test tool to check if WebDAV (a parallel authoring system) is set up correctly, and (3) *JSch,* a graphical front end to ssh and scp. These case studies show the practical usefulness of our work; the last case study shows a scenario in which checkpointing technology is needed.

In our first case study we confirmed several flaws in jget [12]. One defect related to concurrency may prevent a download from completing normally. Likewise, the WebDAV test client has a defect that causes it to crash instead of a shutting down gracefully on a timeout [31].

Our most recent case study found a fault in *ScpTo* [8]. ScpTo is an example program in the *Java Secure Channel (JSch)* package [32], which copies a local file to a remote host via a secure channel. ScpTo contained code that was not supported by JPF such as GUI and a cryptographic library, so we had to abstract some part of ScpTo before running it in the verification system. This program was used as a client, which interacted with an ssh server in our experiment. Both programs produced non-deterministic outputs due to the process of building a secret shared key [33]. Cache-based model checking with DMTCP found a race condition that caused an error in this system. The main thread and a worker thread in ScpTo were not properly synchronized, so there was a case where the main thread could not access an essential data produced by the worker thread, resulting in an exception.

8 Related Work

Model checking automates the correctness proof of a system specification by computing the reachable states of a system [4]. Most model checkers take a specification in their own domain-specific language as input [34], [35]. In addition to labelled transition system with input/output [16], distributed systems are often modelled by a process algebra called π-calculus [36]. PIPER [37] takes another approach to define a type system using Calculus of Communicating Systems (CCS) processes, and SPIN [34] is used to show a relationship between CCS processes.

More recently, model checkers verifying software code have become more prevalent. These software model checkers either verify application code directly [6], [38], [39], [40], [41], or use automatic program abstraction [5], [42], [43] or information collected at run-time [44], [45] to generate a model of the program or its behavior. Manual model construction is not necessary, and thus no modeling language is used as an input language.

The remainder of this section presents software model-checking techniques used in the verification of distributed systems. The verification of distributed systems is not straightforward. One of the challenges is to systematically control interactions and interleavings between processes. Some widely used software model checkers are not capable of manipulating multiple processes directly. Several techniques have been devised to address this shortcoming.

8.1 Scrapbook

This approach implements a model checker for multi-process systems on top of a virtualized environment [46], where the entire system state can be captured into a checkpoint. The tool called *ScrapBook*, an extension of *User-mode Linux* [47], can be used as a building block for implement such a model checker. It adds the checkpointing functionality to User-mode Linux. A model checker can use this function to save the system state when the SUT progresses, and restore the corresponding system state when the SUT backtracks. The downside of the multi-process model checker is that the SUT is suspended by gdb [48]. Therefore, breakpoints must be set manually by a user in advance, discouraging automation. Furthermore, the size of each checkpoint is very large as it contains the information of the entire operating system. Because of this, the size of applications that can be verified is very limited. Other than the purpose of model checking, SBUML is intended to be applied in intrusion detection and sandboxing for network security. This work predates our attempt to use virtualization technology for software verification.

8.2 Centralization

A solution that can use single-process model checkers for distributed systems, is to transform every process to a thread and combine the transformed software into a single process. This process can then be verified by a single-process model checker like Java PathFinder [10], [11]. This approach is called *centralization*. The product of the transformation is a *centralized process*. The centralized process starts all threads converted from the original processes. The threads are running in parallel in the model checker, simulating multiple processes executing concurrently. An model library is also provided to simulate inter-process communication by inter-thread communication. While centralization is a generic solution, the centralized process is usually too complex to be explored exhaustively, since the interleaving among centralized processes increases the state space considerably [12].

8.3 NetStub

Another way to deal with multiple processes is to create a model of the environment that is compatible with the model checker. *NetStub* is a framework that helps testers in building the model for verification [9]. It separates the behavior of distributed systems from the behavior of the network by replacing the Java network library with stub classes. The stub classes simulate the network behavior in order to verify a single process. NetStub models a network by a simple module in order to reduce the size of the state space and focus on the internal function of the system. This approach requires some effort to model the environment, especially when precise behavior is needed for verification. Therefore, it is not fully automated.

8.4 CHESS

Alternatives to software model checking are provided by approaches that discard full state space exploration in principle, and attempt to find defects by a partial analysis instead. *CHESS* [49] is a runtime verification tool that finds faults caused by concurrency. It takes a complete control over thread scheduling and asynchronous events of that program. In other words, all thread interleaving is managed by the tool.

CHESS enhances software testing by executing a test case repeatedly to find concurrent failures. The program state space is gradually covered until the tool reaches the specified number of runs. CHESS applies several techniques to reduce the number of thread schedules in consideration. First, it bounds the number of preemptions when it enumerates thread schedules. The idea behind is that most concurrency bugs happen with only a few preemptions [50]. The verification technique that runs a system under many thread schedules was earlier adopted by *VeriSoft* [39]. CHESS further bounds the search scope by only inserting preemptions into code regions of interest, excluding base libraries that are assumed to be thread-safe. This tool is effective in finding bugs during development, although it does not cover every possible program behavior like software model checkers do.

9 Conclusion and Future Work

Distributed systems are complex, owing to their non-deterministic elements such as the interleaving between processes and threads. Faults in such systems are hard to detect and reproduce. Software model checking is a powerful technique to find faults in a concurrent system by exploring every possible execution path of the system. Most model checkers that directly verify the implementation of a system only support a single process at a time. This article has presented a number of approaches that verify a single process (the SUT), which communicates with other peer processes.

The key problem in the verification of networked software is that the state of the SUT is reverted (backtracked) by a model checker during verification, but the states of the peers are not. A synchronization mechanism is needed to maintain the consistency of the system. Two approaches have been presented: peer restart and peer state capture. The former restarts the peer from the beginning and replays a communication trace to recover system consistency. The peer state capture approach takes a snapshot of the peer in each state and stores it in a checkpoint. The checkpoint can be used to restore the peer in the state corresponding to the SUT.

To improve the performance of verification, cache-based model checking has been presented. It makes use of a cache for capturing communication traces between the SUT and its peers. The cache uses this information to interact with the SUT after state backtracking whenever possible. This reduces the number of peer restart or checkpoint restoration actions, depending on the synchronization approach. Cache-based model checking can be implemented on both synchronization mechanisms. More optimization is possible by omitting unnecessary checkpoints. A checkpointing strategy gives a condition on when to create checkpoints.

Cache-based model checking is implemented as an extension of Java PathFinder (JPF), a Java model checker. A number of model classes for the network library has been written to support network communication between processes. DMTCP is used as a checkpointing tool that captures the states of peer processes. This technique has succeeded in verifying a variety of distributed applications.

The current cache-based approach is limited to applications in a client-server architecture. Each process must take either the role of a client or a server, but not both. This assumption does not hold for peer-to-peer (P2P) applications, which maintain many connections at a time, and where the order of messages usually affects their behavior. Another restriction is the fact that the cache-based approach may miss faults in an SUT that exhibits trace convergence, and for peers where the output depends on internal non-determinism that cannot be seen or controlled by our tool. As our approach only takes the SUT state, not peer states, into account, it does not check how the SUT responds to every input trace from peers.

Future work includes extending the capabilities of our approach and applying cache-based model checking to other application architectures such as P2P. Another possible improvement is to use a model checker for running peers as well as SUT so that low-level non-determinism such as thread scheduling is in control. By doing this, we could analyze the peer behaviors in more detail and selectively perform the ones that potentially reveal faults in the SUT. Scalability limitations may require heuristics to lead the verification to where a fault is likely to occur. Future work also includes studying such heuristics.

Acknowledgments

This work was supported by *kaken-hi* grants 23240003 and 23300004. The authors would like to thank Eric Platon, Richard Potter, and Franz Weitl for their comments on the paper.

REFERENCES

[1] S. Ghosh, *Distributed Systems: an Algorithmic Approach*. Boston: Twayne Publishers, 2006.

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

[3] G. J. Myers, *The art of software testing*. New York : Wiley, 1979.

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

[5] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," 2003.

[6] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering Journal*, vol. 10, no. 2, pp. 203–232, 2003.

[7] Y. Yang, X. Chen, G. Gopalakrishnan, and R. Kirby, "Efficient stateful dynamic partial order reduction," in *Model Checking Software*, ser. Lecture Notes in Computer Science, K. Havelund, R. Majumdar, and J. Palsberg, Eds. Springer Berlin / Heidelberg, 2008, vol. 5156, pp. 288–305.

[8] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model checking distributed systems by combining caching and process checkpointing," in *Proc. 2011 IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, 2011.

[9] E. D. Barlas and T. Bultan, "NetStub: A framework for verification of distributed Java applications," in *Automated Software Engineering Conf.*, Georgia, USA, 2007, pp. 24–33.

[10] S. D. Stoller and Y. A. Liu, "Transformations for model checking distributed Java programs," in *SPIN '01: Proc. 8th international SPIN workshop on Model checking of software*. NY, USA: Springer-Verlag New York, Inc., 2001, pp. 192–199.

[11] C. Artho and P. Garoche, "Accurate centralization for applying model checking on networked applications," in *Proc. 2006 IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006, pp. 177–188.

[12] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe, "Efficient model checking of networked applications," in *Proc. TOOLS EUROPE 2008*, ser. LNBIP, vol. 19. Zurich, Switzerland: Springer, 2008, pp. 22–40.

[13] 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. 2009 IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2009)*. Washington, DC, USA: IEEE Computer Society, November 2009, pp. 447–458.

[14] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, pp. 375–408, September 2002.

[15] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *13th Int. Conf. on Computer Aided Verification (CAV 2001)*, ser. LNCS, vol. 2102. Paris, France: Springer, 2001, pp. 260–264.

[16] J. Tretmans, "Model based testing with labelled transition systems," in *Formal Methods and Testing (FORTEST 2008)*, ser. Lecture Notes in Computer Science, vol. 4949. Springer, 2008, pp. 1–38.

[17] M. Rieker and J. Ansel, "Transparent user-level checkpointing for the native POSIX thread library for Linux," in *Proc. of The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2006, pp. 492–498.

[18] P.-H. Kamp, "The one-second war," *Commun. ACM*, vol. 54, pp. 44–48, May 2011.

[19] J. Stultz, "Potential fix for leapsecond caused futex issue," 2012. [Online]. Available: https://lkml.org/lkml/2012/7/1/203

[20] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[21] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Prentice Hall PTR, 1999.

[22] W. Leungwattanakit and C. Artho, *Project net-iocache*, 2012. [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/net-iocache

[23] NASA Ames Research Center, *JPF Developer Guide*, 2012. [Online]. Available: http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/start

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

[25] K. Kim, T. Yavuz-Kahveci, and B. Sanders, "JRF-E: using model checking to give advice on eliminating memory model-related bugs," *Automated Software Engineering*, vol. 19, no. 4, pp. 491–530, 2012.

[26] Red Hat, Inc., "KVM," http://www.linux-kvm.org.

[27] "OpenVZ documentation," http://wiki.openvz.org.

[28] Oracle, "Virtualbox," http://www.virtualbox.org/.

[29] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: transparent checkpointing for cluster computations and the desktop," in *Proc. 2009 IEEE International Symposium on Parallel & Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[30] ACME Laboratories, "thttpd - tiny/turbo/throttling HTTP server," http://www.acme.com/software/thttpd/.

[31] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Verifying networked programs using a model checker extension," in *ICSE Companion proceedings*, Vancouver, Canada, 2009, pp. 409–410.

[32] JCraft, Inc., "JSch - Java Secure Channel," http://www.jcraft.com/jsch/.

[33] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, November 1976.

[34] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[35] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell." *STTT*, pp. 134–152, 1997.

[36] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, i," *Information and computation*, vol. 100, no. 1, pp. 1–40, 1992.

[37] S. Chaki, S. K. Rajamani, and J. Rehof, "Types as models: model checking message-passing programs," *SIGPLAN Not.*, vol. 37, no. 1, pp. 45–57, Jan. 2002.

[38] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimuller, "JNuke: Efficient dynamic analysis for Java," in *Proc. 16th Int. Conf. on Computer Aided Verification (CAV 2004)*, ser. Lecture Notes in Computer Science, R. Alur and D. Peled, Eds., vol. 3114. Springer, 2004, pp. 462–465.

[39] P. Godefroid, "Software model checking: The VeriSoft approach," *Form. Methods Syst. Des.*, vol. 26, no. 2, pp. 77–101, 2005.

[40] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," in *Proc. Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[41] Y. Yang, X. Chen, and G. Gopalakrishnan, "Inspect: A runtime model checker for multithreaded C programs," University of Utah, USA, Tech. Rep., 2008.

[42] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *Proceedings of the 8th international SPIN workshop on Model checking of software*, ser. SPIN '01. New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 103–122.

[43] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from java source code," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, 2000, pp. 439 –448.

[44] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 265–278.

[45] Z. Qi, L. Liu, A. Liang, H. Wang, and Y. Chen, "An online model checking tool for safety and liveness bugs," in *Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on*, dec. 2008, pp. 493–500.

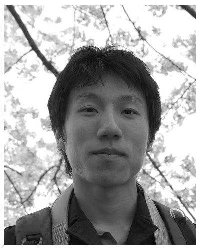
[46] Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato, "Model checking of multi-process applications using SBUML and GDB," in *Workshop on Dependable Software: Tools and Methods*, Yokohama, Japan, 2005, pp. 215–220.

[47] J. Dike, *User Mode Linux*. Prentice Hall PTR, 2006.

[48] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB : the GNU source-level debugger*, 9th ed. Boston, MA : Free Software Foundation, 2002.

[49] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proc. 8th USENIX conference on Operating systems design and implementation (OSDI 2008)*. Berkeley, CA, USA: USENIX Association, 2008, pp. 267–280.

[50] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *SIGPLAN Not.*, vol. 43, no. 3, pp. 329–339, Mar. 2008.



Watcharin Leungwattanakit received the MSc and PhD degrees from the University of Tokyo in 2008 and 2011, respectively. He was a post-doctoral researcher at the University of Tokyo and Chiba University in 2012. Currently, he is a senior technical consultant at VenTek International (Thailand). His interests include software model checking, testing, and enterprise search technologies.

Cyrille Artho 's main interests are software verification and software engineering. In his Master's thesis, he compared different approaches for finding faults in multi-threaded programs. Later in his Ph.D. thesis, he continued his search for such defects, earning his Doctorate at ETH Zurich. During that research, he spent two summers at the Computational Sciences Division of the NASA Ames Research Center. After graduation he worked at NII, Tokyo, for two years, and then moved to AIST. He currently holds the position of Senior Researcher at AIST Amagasaki, Japan. His most recent work covers the analysis of networked systems, using software model checking and test case generation.

Masami Hagiya received MSc from Department of Information Science, University of Tokyo in 1982, and PhD from Research Institute for Mathematical Sciences, Kyoto University in 1988. He is currently a professor in Computer Science, University of Tokyo. With background in formal logic, he is interested in analysis, verification and synthesis of computational models in general. He has been working on formal verification of computing systems, recently on model checking of networked software by extending Java PathFinder. He also has been interested in applying computational models to biological and molecular systems. Beginning with research on DNA computing, he has been leading research on molecular computing and DNA nanotechnology in Japan, and one of his current research projects is on molecular robotics. Finally, he is also working on IT education at the levels of high school and university general education with the hope of raising IT literacy of the Japanese society.

Yoshinori Tanabe received the MSc degree in Mathematics from Tsukuba University in 1987. After working for Fujitsu and other companies for about ten years, he received the PhD degree in Information Science And Technology from the University of Tokyo in 2008. He is currenty Research Professor at the National Institute of Informatics, Tokyo, and interested in software verification.

Mitsuharu Yamamoto received the MSc and PhD degrees from the University of Tokyo in 1996 and 2003, respectively. He is an associate professor in the Department of Mathematics and Informatics at Chiba University. His research interests include formal verification, model checking, and theorem proving.

Koichi Takahashi received his B.S.and M.S. degrees in mathematics from Nagoya University in 1986 and 1988, and Ph.D. degree in information science from the University of Tokyo in 2002. Since 1988 he has worked at the Electrotechnical Laboratory (currently the National Institute of Advanced Industrial Science and Technology). His research interests include theoretical verifications.