SAMC: A Fast Model Checker for Finding Heisenbugs in Distributed Systems (Demo)

Tanakorn Leesatapornwongsa
University of Chicago, USA
tanakorn@cs.uchicago.edu

Haryadi S. Gunawi
University of Chicago, USA
haryadi@cs.uchicago.edu

ABSTRACT

We present SAMC, an open-source model checker that can be integrated to many modern distributed cloud systems. SAMC can find concurrency bugs caused by non-deterministic distributed events. We have successfully integrated SAMC to Hadoop, ZooKeeper and Cassandra.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*

General Terms

Reliability, Verification

Keywords

Concurrency bugs, model checking, distributed systems

1. INTRODUCTION

The past eight years have seen a rise of distributed system model checker ("dmck" in short) for verifying the reliability of distributed systems [2, 3, 6, 8, 12, 13, 14, 15]. A dmck is a software (implementation-level) model checker targeted for distributed systems. It works by exercising all possible sequences of events (*e.g.*, different reorderings of messages), and hereby pushing the target system into corner-case situations and unearthing hard-to-find bugs. To be highly practical in checking large-scale systems, a dmck must address the state-space explosion problem, for example, by adopting advanced state reduction techniques (dynamic partial order reduction, symmetry, etc.) [6, 11, 15].

Behind this trend, in the past few years, there is a surge of adoption of scalable distributed systems ("cloud systems" in short) such as distributed parallel computing frameworks, key-value stores, file systems, and synchronization services. Some of the open-source cloud systems including Hadoop, HBase, Cassandra, HDFS, and ZooKeeper, have been adopted massively. In our group, we have performed an in-depth study of thousands of reported bugs in these systems and we find that complex distributed concurrency bugs are quite common and continue to appear throughout the development process [5, 10, 11]. However, we did not find a single available dmck that can be adopted to modern distributed cloud systems. For example, MODIST [15], an OS-interposed dmck is proprietary and not open-sourced, and its implementation also has adoptability issues. MACEMC [8], a popular dmck in the academic world, only works on systems built on Mace domain-specific language. (We elaborate these issues later in Section 3). This situation forced us to create a new adoptable dmck.

In this paper, we present Semantic-Aware Model Checker (SAMC), an open-source dmck that can be adopted to many modern distributed cloud systems [1]. We built SAMC (pronounced "Sam-C") from scratch for a total of over 10 KLOC. We built it in a completely different way than existing approaches (§3). We successfully integrated SAMC to Hadoop, ZooKeeper and Cassandra. In our recent work, we utilized SAMC to invent several semantic-aware exploration algorithms [11]. In this demonstration paper, our focus is about the implementation and integrations details of SAMC. In the following sections, we describe our design decisions behind SAMC (§3) along with implementation and integration details (§4).

2. DMCK PRIMER

We first describe the general anatomy of a dmck. As defined above, dmck is a software model checker that checks distributed systems directly at the implementation level. That is, dmck re-orders distributed events as the system runs. Figure 1 illustrates a dmck integration to a target distributed system (*e.g.*, a 2-node system). The dmck inserts an *interposition layer* (the gray box) in each node in the target system with the purpose of controlling all non-deterministic distributed events (*e.g.*, asynchronous network messages, timeouts). Thus, instead of letting the nodes execute these events non-deterministically, the interposition layer intercepts the events and let the *dmck server* decide which events should be enabled and in what order (*i.e.*, the ordering permutation). For example, the figure shows that there are four outstanding events (*e.g.*, `abcd`) and the dmck server decides to `enable(b)` first. In checking a real system, a dmck typically generates thousands of executions; an *execution* (or a *path*) is a specific ordering of events that the dmck enables from an initial state to a termination point

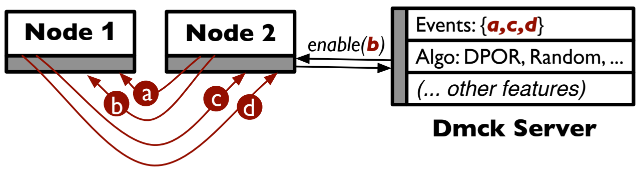

Figure 1: A Dmck Architecture.

(*e.g.*, `abcd`, `dbca`). To permute events, the dmck server can use different *exploration algorithms* from simple ones such as depth first search and random to advanced ones such as partial-order reduction and symmetry.

To model check a specific distributed protocol (*e.g.*, leader election), dmck starts the *workload driver* (not shown in the figure) which runs a specific workload (*e.g.*, node joins/departures, file read/write), restarts the whole system after every execution, and stops when the dmck server has exercised all possible executions. As events are permuted, the target system enters hard-to-reach states. Dmck continuously runs *safety checks* on the global state (*e.g.*, "there should be no two leaders") and local states via per-node assertions (*e.g.*, "a read should return the latest version").

In building a complete dmck, implementing the server side is relatively straightforward; the server is a single centralized process that communicates with all interposition instances. The major challenges are: integrating the interposition layer to a target system, intercepting the distributed events that must be permuted, and enabling powerful exploration algorithms at the dmck server.

3. DMCK FAMILY

In this section, we describe a family of dmck implementations proposed in the literature. To the best of our knowledge, there are only three categories of dmck: OS-, application-, and DSL-interposed dmck. We discuss the pros and cons of each category using four characteristics: transparency, adoptability, extensibility, and versatility (Table 1). These metrics answer the following questions. *Transparency:* Can the dmck test a new target system without any engineering effort? *Extensibility:* How easy is it to incorporate new types of events that can be permuted by the dmck? *Versatility:* Can the dmck support the use of "rich" model checking techniques (partial-order reduction, symmetry, semantic-based, etc.)? *Adoptability:* Can the dmck be adopted to many other distributed systems?

3.1 OS/Library-Interposed Dmck

There are several OS/library-interposed (lib-interposed) dmcks that have been proposed. [2, 12, 15]. A lib-interposed dmck intercepts distributed events at the system call or library level (*e.g.*, network packet send and receive).

Pros: *Transparency:* The advantage of lib-interposed dmck is in its transparency. MODIST [15] for example is built around Windows OS; distributed systems built on Windows (that uses WinAPI specifically) can be transparently model checked without much engineering effort. Due to its transparency, lib-interposed dmck is capable of model checking real-world distributed systems (*e.g.*, the success of MODIST in checking many distributed systems inside Microsoft).

Cons: *Adoptability:* Lib-interposed dmck is constrained to a particular OS/library and its APIs (*e.g.*, MODIST cannot help systems using C POSIX API or Java SDK libraries). *Extensibility:* Because the interposition is restricted at the system/library call level (for the sake of transparency), lib-interposed dmck is hard to extend. Incorporating and permuting new application events (*e.g.*, timeouts) is challenging as developers can implement timeout mechanisms in many different ways. MODIST must perform custom source code analysis integrated with the system-call interpositioning. Extending this requires further changes at the OS level. *Versatility:* Lib-interposed dmck cannot analyze event content because an event is merely a stream of bytes. For example, the application-specific structures of messages are lost. This impedes the use of advanced model-checking techniques (*e.g.*, partial-order reduction, symmetry) that require "white-box" knowledge [11].

Table 1: DMCK Comparision

	Interposition level:		
	OS/Lib	Application	DSL
Transparent	✓	✗	✓
Adoptable	✗	✓	✗
Extensible	✗	✓	✓
Versatile	✗	✓	✓

3.2 Application-Interposed Dmck

Application-interposed dmck interposes at application or library-call functions. The tester first decides what events to permute and then writes wrappers that intercept and forward the events to the dmck server.

Pros: *Adoptability:* Application-level interposition automatically implies adoptability. The only drawback is its non-transparency. However, we emphasize that the non-transparent part is *only* at the interpositioning side. The permutation mechanisms and model-checking algorithms implemented at the dmck server side can *stay the same.* We discuss this more below. *Extensibility:* Application-level interpositioning is extensible because the tester can easily write wrappers for application routines (*e.g.*, timeout mechanisms) and the dmck server will permute the new events. *Versatility:* By interpositioning at application-specific functions, the white-box information (*e.g.*, message structures, timeout values) is not lost. At the dmck server, the tester can write powerful model checking algorithms that use white-box knowledge [11]. We show later that white-box information can be encapsulated as a set of key-value pairs (§4).

Cons: *Transparency:* The drawback of app-interposed dmck is its non-transparency. The tester must decide what functions to interpose, write wrappers around those functions to notify the dmck server about the events.

Given the pros and cons above, we believe that application-interposed dmck is the right approach for current and future distributed systems. Specifically, we argue that *transparency is an unnecessary dmck feature.* There are several reasons for this. First, today's developers tend to be the testers and vice versa, especially for complex cloud distributed systems. We believe it is easy for the developers (the experts of the target system) to implement the interposition layer (wrappers) and modify the original code slightly whenever necessary. Second, this process can be done non-intrusively with mature interpositioning technologies (*e.g.*, AspectJ for Java). That is, the target system code will not be cluttered with testing code. Third, the non-transparency is only in the interposi-

Table 2: SAMC Code Size.

	Common (re-usable)	SampleSys-specific	ZooKeeper-specific
Server	7519	17	86
Workload driver	274	30	206
Wrapper	31	18	217

tion side. To integrate an application-interposed dmck (*e.g.*, in Java) to a new target system (*e.g.*, in C), the tester can simply write wrappers on the target system side (in C). However, *the dmck server* (where the permutation mechanisms and model-checking algorithms live) *can stay the same*. The interposition layer and the dmck server can communicate in language-agnostic manner using a client-server network library. Fourth, although there is room for human error (*e.g.*, important events are missed), this problem is fixable; we believe the tester can progressively ensure that all important events are covered. In summary, advantages in adoptability, extensibility, and versatility are much more powerful, and sacrificing transparency can be justified.

Besides SAMC, we are aware of one other application-interposed dmck, dBug [13], which targets C language and requires manual insertion of interceptors (*e.g.*, does not use AspectC). Compared to dBug, SAMC has a richer interface that empowers white-box techniques. SAMC also has a replay tool.

3.3 DSL-Interposed Dmck

The last category is dmck that is integrated tightly with a domain-specific language (DSL). A popular example is MaceMC [8] from the Mace language development tool [9]. Mace is a DSL for building distributed systems.

Pros: *Transparency:* No engineering effort is required to insert an interpositioning layer; when using the language, developers declare distributed events, which will automatically be intercepted by MaceMC. DSL-interposed dmck is transparent only to systems written in the DSL. *Extensibility:* Permuting new types of events can be done by modifying the DSL compiler to define additional domain-specific events to permute. *Versatility:* By intercepting events at the DSL level, the model checker can analyze the events as the application-specific information is not lost. Compared to application-interposed dmck which interposes "mainstream" programming languages, DSL-interposed dmck is harder to extend; only few developers understand specific DSL.

Cons: *Adoptability:* The biggest drawback of DSL-level interpositioning is its adoptability. If the DSL is not popular, then the included dmck is of little use in practice. Despite of MaceMC popularity in the research community (as it is open sourced) we are not aware of real deployed distributed systems that use Mace language. Thus, techniques implemented in MaceMC [3, 6]) must be re-implemented from scratch to be integrated to other systems.

4. SAMC AND INTEGRATION

There are three important parts of SAMC: the interpositioning layer, the dmck server side (including the interface between the two) and the workload driver. Table 2 shows the code size of the common part of SAMC, the simple integration (*e.g.*, SampleSys with a minimum requirement and default exploration; SampleSys is described in Appendix), and an advanced one (*e.g.*, integration to ZooKeeper with advanced white-box DPOR and symmetry algorithms [11]). Currently, SAMC can be easily integrated to distributed systems written in Java. We target Java because many cloud distributed systems under the Apache Software Foundation (ASF) are written in Java. Non-Java target systems require interpositioning methods in their corresponding programming languages.

```
 1 pointcut sendMsg(Msg m) :
 2   call(public void *.sendMsg(Msg)) && args(v);
 3
 4 void around(Msg m) : sendMsg(m) {
 5    Event e = new Event();
 6    e.addKV ('eventId', String.hash(m.toString));
 7    e.addKV ('sender', m.sender);
 8    e.addKV ('receiver', m.receiver);
 9    e.addKV ('vote', m.vote);
10    e.addKV ('myVote', s.myVote);
11    dmckServer.addEvent(e);
12    proceed();
13 }
```

Figure 2: Interpositioning.

4.1 Interpositioning

To interpose SAMC to a target system, a tester performs the following steps. First, the tester identifies important events (*e.g.*, send message, disk read/write) whose timing should be controlled; an important event is usually an application-specific function, for example, `sendMsg(Msg M)`. Second, the tester weaves the function call with a simple *pointcut* in AspectJ (an AOP tool for Java) as shown in line 1-2 in Figure 2. AspectJ is used for not cluttering the target system with interpositioning code. The "makefile" should be modified as well to include aspect code. We provide pointcut templates for novice users. Third, the tester converts application-specific events (*e.g.*, `Msg`) into a generic structure to be sent via the server interface.

4.2 Event Key-Value Interface

The communication interface between the interposition layer and the dmck server must stay the same for different target systems. Thus, we use an abstraction `Event`, which is a set of key-value pairs. The tester can define *any* key-value pairs. The *minimum* requirement is the `eventId` key whose value is a unique event ID. For example, in line 6 in Figure 2, we set the `eventId` as the hash value of `Msg m`. (In Section 2, `a`, `b`, `c`, and `d` can be considered as an event ID). The dmck server will use event IDs to remember the permutation/exploration history (thus, an event ID cannot be a random, non-deterministic, or a simplistic ID).

If the tester wants to use white-box information about the event for faster state-space exploration, the tester can incorporates more information in the `Event` abstraction. For example, in line 7-10 in Figure 2, the tester incorporates more application-specific information such as the `sender` and `receiver` nodes and the `vote` number.

After `Event` is populated, the tester can easily send it to the server using an RPC (Java RMI in line 11; can be extended to language-agnostic client-server interface). At this point, the event in the target code (*i.e.*, `sendMsg(m)`) is *blocked* and cannot continue until the RPC returns (*i.e.*, until the dmck server enables the event).

4.3 Dmck Server

The server side maintains a *queue* of outstanding events. The exploration algorithm decides which events should be

```
1 while (!noNewPath()) {
2     cleanAndSetup();
3     runProtocol();
4     checkResult(); // can stop here
5 }
```

Figure 3: Workload Driver.

enabled (by returning from the RPC call). By default, SAMC runs a depth-first search (DFS) or Random algorithms. These algorithms do not need white-box information and thus can work solely based on `eventID` (*e.g.*, permute event IDs `a`, `b`, `c`, and so on). For DFS and Random, the tester does not need to add new code to the server.

We also provide more advanced algorithms such as dynamic partial order reduction (DPOR) and symmetry [11]. Let's consider creating a DPOR algorithm: "a message to a given node is independent of other concurrent messages destined to other nodes" [13, 15]. This algorithm requires information about the message `receiver` (which we already collected above). At the server side, the tester can implement this algorithm by using `event.getValue('receiver')`.

Our framework empowers testers to develop new exploration algorithms. The goal of an exploration algorithm is to decide which event `e` to enable. Here, `e` is essentially a function of (1) the queue of outstanding events, (2) the permutation history, and (3) the state of the target system. That is, `e=func(queue,history,state);`. In our framework, (1) and (2) are available by default. The system state (3) is application specific and can be "piggy-backed" with event interpositioning (*e.g.*, a simple `myVote` local state in line 10 in Figure 2). There are other ways to pass local states to the server (out of the scope of the paper).

After creating a custom exploration (`func`), the tester can enable the chosen event by calling `enable(e)`. After this, the server waits for one second (configurable) to let the target system quiesces before the server enables a new event. That is, as an event is enabled, the target system might generate more events (a reaction) and the dmck server must wait for the new events to be recorded. A faster wait time is possible but requires more modifications in the target code (out of the scope of this paper).

To uncover deep bugs, our server can also enable *failure* events (*e.g.*, crashes); bugs often appear in recovery paths [4, 7, 11]. Failure events do not originate from the interposition layer. The tester must specify at the server side the specific command lines to kill a particular node/process.

4.4 Workload Driver and Replay

The workload driver contains a few simple steps as shown in Figure 3. The driver communicates with the dmck server to check if all paths (based on the chosen algorithm) have been explored (`noNewPath()`). The rest of the steps are application specific; the testers must specify how to clean and setup the target system (line 2), run the distributed protocol to be tested (line 3), and check the result of the execution (line 4). An iteration of the while loop represents an execution (a path) of a sequence of events (*e.g.*, `acbd`; §2).

For example, to test a distributed commit protocol, the `cleanAndSetup()` starts the system from scratch and prepare a preliminary data, the `runProtocol()` contains code that has concurrent updates to different nodes in the target system, and the `checkResult()` searches assertion violations in the log files maintaned by each node in the target system. Since a log file is a local view only, the tester can also deploy global checks at the dmck server side that the `checkResult()` can check.

If an error (bug) is found, the path (sequence of events) to the error is recorded by the dmck server. The tester can also save the log files. SAMC also provides a replay tool (not shown); the tester can feed SAMC the buggy path and SAMC can deterministically replay the path.

5. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their tremendous feedback and comments. This material is based upon work supported by the NSF (grant Nos. CCF-1321958, CCF-1336580, and CNS-1350499) and generous supports from NetApp.

6. REFERENCES

[1] `http://ucare.cs.uchicago.edu/projects/samc/`.

[2] Elliot D. Barlas and Tevfik Bultan. NetStub: A Framework for Verification of Distributed Java Applications. In *ASE '07*.

[3] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *NSDI '11*.

[4] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI '11*.

[5] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *SoCC '14*.

[6] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP '11*.

[7] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *OOPSLA '11*.

[8] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI '07*.

[9] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language support for building distributed systems. In *PLDI '07*.

[10] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. The Case for Drill-Ready Cloud Computing. In *SoCC '14*.

[11] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI '14*.

[12] Watcharin Leungwattanakit, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, Mitsuharu Yamamoto, and Koichi Takahashi. Modular Software Model Checking for Distributed Systems. *IEEE Transactions on Software Engineering*, 40(5):483–501, May 2014.

[13] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *SSV '10*.

[14] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI '09*.

[15] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09*.