Analyzing Distributed Java Applications by Automatic Centralization

Lei Ma
Graduate School of Engineering
The University of Tokyo
malei@satolab.itc.u-tokyo.ac.jp

Cyrille Artho
Research Institute for Secure Systems
AIST
c.artho@aist.go.jp

Hiroyuki Sato
Information Technology Center
The University of Tokyo
schuko@satolab.itc.u-tokyo.ac.jp

***Abstract*—The verification and analysis of distributed applications are difficult. They involve large combinational states, interactive network communication between peers, and concurrency. Some dynamic analysis tools can analyze the runtime behavior of a single-process application. However, they do not support the analysis of a whole distributed application, where multiple processes run simultaneously.**

Centralization is a general solution, which transforms multi-process applications into a single-process one that can be directly analyzed by such existing tools. In this paper, we adopt centralization as a general framework for analyzing distributed applications. We propose and solve the essential issue of a class version conflict during centralization. We also propose a clean solution for the shutdown semantics. We implement and apply our centralization tool to some network benchmarks. Experiments, where existing tools are used on the centralized application, support the usefulness of our automatic centralization tool. Centralization enables existing single-process tools to analyze distributed applications.

***Keywords*-Distributed application; dynamic analysis; software model checking**

I. Introduction

Analyzing distributed applications is challenging. Multiple processes run concurrently and use asynchronous communication over a network. Activities of processes can be arbitrarily interleaved and no two executions of the same application need be the same. Such nondeterminism causes the run-time behavior of distributed applications to be difficult to predict, debug, and verify. This problem gets exacerbated if multiple threads inside a process are involved, creating concurrency inside a process as well as between processes. Most non-trivial applications nowadays are implemented as distributed, networked applications where multiple processes are combined into a complex system. Analysis and verification of such distributed applications are therefore very important. Some existing tools like Java PathFinder (JPF) [12], Java Runtime Analysis Toolkit (JRAT) [7] work on single-process applications, but they do not support multi-process applications. If powerful analysis tools that support a single process were available for multiple processes, development and analysis of distributed systems would become easier.

Process *centralization* [11], [1] is a solution to enable existing tools to analyze multi-process applications. It transforms a multi-process application into a single one with the equivalent runtime behavior. Fig. 1 shows the centralization of a distributed application containing three components: one server and two clients. Before centralization, each component runs as a process. Inside the server process, three threads run concurrently. Thread $main$ creates two $Worker$ threads to serve each connected client, separately. After centralization, all processes are wrapped as threads and run as one process.

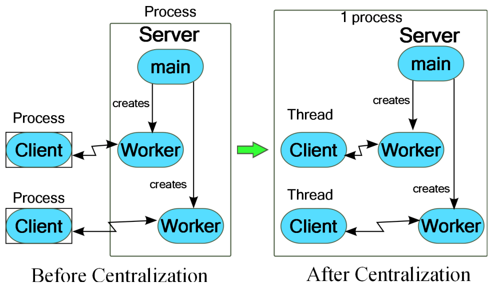


Figure 1: Process Centralization Example

Centralization was initially proposed to verify distributed applications exhaustively. However, the large combinational states limit such an analysis to small applications. We propose to use centralization for general (not necessarily exhaustive) analysis of distributed applications. Centralization enables many existing tools for integrated analysis and reduces the difficulty of analyzing distributed applications. For example, in a single-process debugger, the entire centralized system cannot be paused simultaneously; when centralized this becomes possible. Some dynamic verification tools such as JCarder [8] detect deadlock bugs for single-process applications, but they do not support multi-process applications. Meanwhile, profiling tools [7] are useful for analyzing the runtime performance of distributed applications. As a centralized application runs on a single VM, centralization enable these tools for integrated profiling of the whole distributed application.

There are currently no automatic centralization tools available. A previous centralization tool [1] is outdated and unable to work on current Java applications. Previous work mainly used centralization to verify multi-process applications with JPF [12]. Certain aspects of the implementation such as system startup and shutdown are targeted to JPF and not applicable to other analysis tools. When moving beyond JPF, larger systems can be supported, making tool automation all the more important.

Furthermore, one essential centralization issue, classes with multiple versions in different components, is not covered by previous tools. This is a common occurrence in component-based systems, where different parts are developed independently and thus may use different versions of library classes.

In this paper, we improve centralization to support the general analysis of distributed applications. We propose a general solution to handle classes with multiple versions, and also a solution for the shutdown semantics.

Several existing tools can benefit from our tool for analyzing distributed applications, as demonstrated by experiments. We discuss and verify some network fault models with JPF, allowing some defects to be found that are missed with a single-process analysis. We also discuss how existing profiling tools can be used for analyzing multi-process applications by centralization.

The rest of this paper is organized as follows. Section II summarizes the centralization issues. Section III formalizes the multiple class version issue and explains our solution. Section IV proposes our solution to the shutdown semantics. Section V describes the implementation and experiments using our tool. Section VI presents related work. Section VII concludes and discusses future work.

II. Centralization issues

This section summarizes the problems that have to be solved to the implement centralization of distributed applications correctly.

Definition 1: The term *distributed application* contains three aspects [3]: Firstly, it means an application whose functionality is split into a set of cooperating, interacting components; each component has an internal state and operations on its state. Secondly, these components can be assigned to different machines. Finally, the functional components exchange information through the network.

On modern operating systems, distributed applications are implemented as a system using multiple processes, which usually run on different hosts and communicate over a network. Centralization transforms such a *multi-process* system into a *single-process* one, while preserving the semantics of the combined system. The transformed system runs on a single host, and all communication between the transformed processes is internalized.

This paper is concerned with the centralization of programs written in Java [5], a very popular programming language that is designed to facilitate the creation of networked applications. The concepts presented in this paper generalize to other platforms using threads, shared memory, and inter-process communication, although their implementations may differ. A *centralized program* is the program after centralization. Centralization must preserve the semantics of original program. For each execution in the original program, there exists an execution trace in centralized program with the same behavior, and vice versa. To satisfy this requirement, the following issues must be addressed:

1) Version separation: Multiple versions of a class may occur in different components of a distributed application. Before centralization, each component runs as a process on its own Virtual Machine (VM) and holds its own version of each class locally. Because a centralized program runs on single VM, each class is loaded and defined once. Direct centralization is incorrect, if multiple versions of a class with the same name exist. We propose our solution in Section III.

2) Memory space separation: In a multi-process system, the operating system separates the memory spaces of all processes. This separation is absent in the centralized program but can be emulated by program transformation. In Java-like systems, memory space separation is only necessary on static data, which exists once per VM. Static fields and class descriptors are shared as a singleton instance of a given class. Accessing these data by different processes without proper separation in the centralized program will cause data races. Therefore, centralization should keep the memory space of each process separate [11], [1].

3) Runtime behavior: Startup and shutdown. Centralization wraps each process of original program as a group of threads and starts them as such. We denote each group of such threads as a *centralized process*. For the analysis of network applications, ensuring the server is initialized before clients try to connect is important. Otherwise, a client exits prematurely after failing to connect to the server. Regarding the shutdown semantics of the original program, if a process exits it terminates all its threads. It also releases all the resources like socket ports. Its VM is shut down while other processes might continue running. Centralization should preserve the startup order of each centralized process and the shutdown semantics. A solution given in previous work [1] is specific to JPF. We discuss our general solution in Sections IV and V, respectively.

III. Version separation

The usage of slightly different versions of library classes is common in component-based systems, where each component is developed and managed independently. Centralization is incorrect without properly separating the class namespace for each component. In this section, we formalize and propose our solution to this issue.

A. Class Abstraction and Classification

A Java class can be uniquely identified by its name (including package name) and implementation. For a class cl, we use $cl.name$ and $cl.code$ to denote the class name and its implementation, respectively. Given two classes cl_1 and cl_2, cl_1 is equivalent to cl_2, denoted by $cl_1 = cl_2$, iff both of their names and codes are identical.

Definition 2: A *project* is a set of classes. We denote a class cl in a project p by $p.cl$.

Definition 3: Given a project p, we define $\text{NAME}(p) = \{cl.name | cl \in p\}$ as the set of all class names in p.

A *project* abstracts the implementation of a component in distributed applications. Each process may use code from a different project but code from a given project may also be shared among processes.

Definition 4: Process centralization is the transformation of multiple processes into a single one with equivalent runtime behavior.

Previous work [1] assumes all the processes run under the same project, where each class has only one version. To centralize processes containing classes with multiple versions, we propose to perform *project centralization*. Before defining project centralization, we first define project renaming substitution and project equivalence.

Definition 5: Given a project p, and class names n_1,n_2, project renaming substitution $p[n_1/n_2]$ is defined as a project in which n_1 in p is substituted for n_2. A renaming substitution $p[n_1/n_2]$ is normal iff $n_{1 \notin \text{NAME}(p)}$ and $n_2 \in \text{NAME}(p)$.

Definition 6: Given two projects $p_1$and p_2, $p_1$is equivalent to p_2, denoted by $p_1 = p_2$ iff they can be renamed to the identical projects by normal renaming substitutions.

Definition 7: Project centralization transforms a project set P into one single project p_{centra} such that $\forall p \in P. \exists p' \subseteq p_{centra}.p = p'$.

Project centralization requires preservation of the class namespace and implementation for each project. Each process that runs on one of original projects can also run on the centralized project with the same runtime behavior.

Definition 8: Given two classes cl_1 and cl_2 in a project p, cl_1 depends on cl_2, denoted by $cl_2 \rightarrow cl_1$ if $cl_1.code$ references $cl_2.name$. For a class $cl \in p$, we define $\text{DEPENDS}(cl, p) = \{cl' \in p | cl \rightarrow cl'\}$.

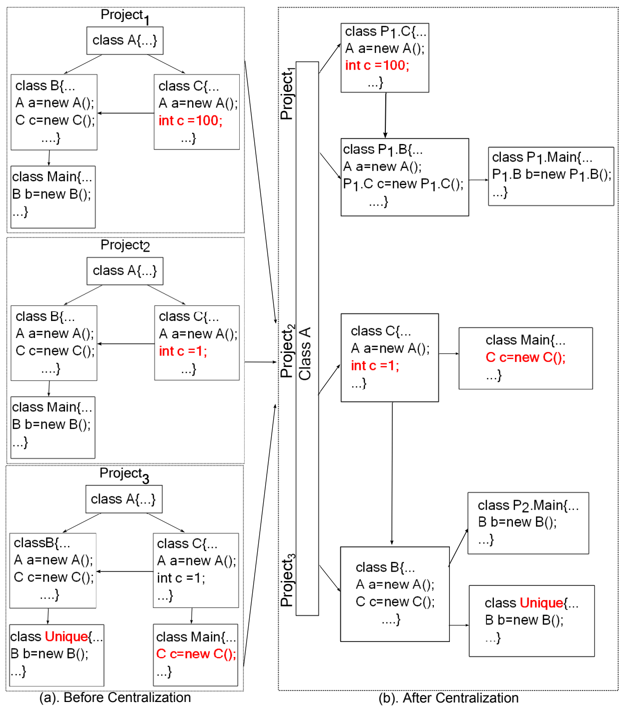
The class dependency represents the class reference relation in a project. $\text{DEPENDS}(cl, p)$ is the set of classes in p that reference cl;

Let P be a set of projects. To separate the class namespace of each project $p \in P$, we classify the classes of p into the following categories:

1).*Unique Class*. $\text{UNIQUE}(p, P) = \{cl \in p | \forall q \in P.(p \not\equiv q \Rightarrow cl.name \notin \text{NAME}(q))$. A unique class of p is the class that has a unique name in p, and this name does not occur in any other projects.

2).*Conflict Class*. $\text{CONFLICT}(p, P) = \{cl \in p | \exists q \in P.(cl.name \in (\text{NAME}(p) \cap \text{NAME}(q)) \wedge p.cl.code \neq q.cl.code\}$. The name of a conflict class in p appears in multiple projects, but with a different implementation.

3).*Shared Class*. $\text{SHARED}(p, P) = \{cl \in p | \exists q \in P.(cl.name \in (\text{NAME}(p) \cap \text{NAME}(q)) \wedge p.cl = q.cl\}$. A shared class of p shares both its name and implementation among multiple projects.



Project$_1$
class A{...}
class B{... A a=new A(); C c=new C(); ....}
class C{... A a=new A(); int c =100; ...}
class Main{... B b=new B(); ...}
Project$_2$
class A{...}
class B{... A a=new A(); C c=new C(); ....}
class C{... A a=new A(); int c =1; ...}
class Main{... B b=new B(); ...}
Project$_3$
class A{...}
classB{... A a=new A(); C c=new C(); ....}
class C{... A a=new A(); int c =1; ...}
class Unique{... B b=new B(); ...}
class Main{... C c=new C(); ...}
(a). Before Centralization
Project$_1$
Project$_2$
Class A
Project$_3$
class P$_1$.C{... A a=new A(); int c =100; ...}
class P$_1$.B{... A a=new A(); P$_1$.C c=new P$_1$.C(); ....}
class P$_1$.Main{... P$_1$.B b=new P$_1$.B(); ...}
class C{... A a=new A(); int c =1; ...}
class Main{... C c=new C(); ...}
class P$_2$.Main{... B b=new B(); ...}
class B{... A a=new A(); C c=new C(); ....}
class Unique{... B b=new B(); ...}
(b). After Centralization

Figure 2: Project Centralization Example

B. Example

Fig. 2.(a) shows an example to centralize three projects, where edges represent the class depedencies. Project$_1$ and project$_2$ share most of the classes except different versions of class C. Compared with project$_2$, project$_3$ holds a different version of class $Main$ and a new class $Unique$. In this example, classes A and B are shared between all projects. Class C is a conflict class in project$_1$, but it is both a conflict and shared class in project$_2$ and project$_3$. Similarly, class $Main$ is a conflict class in project$_3$, and it is both shared and conflict in project$_1$ and project$_2$.

C. Project centralization and class renaming

Consider centralizing processes from the project set $P = \{p_1, p_2, \ldots, p_n\}$, where one or more processes are started from within each project. Direct centralization of these processes is incorrect if $\text{CONFLICT}(p_i, P) \neq \emptyset$. Project centralization resolves such class version conflicts, while preserving the semantics of each project. After project centralization, process centralization is simplified without having conflict classes. We adopt the class renaming approach for project centralization.

A trivial solution would entail renaming all classes, duplicating all code for each project. However, excessive code duplication will consume much runtime memory and storage. For example, when analyzing a distributed system containing 20 peers, duplicating all projects from these peers is not necessary as they can reuse some shared classes with proper transformation. Therefore, it is beneficial to share common class code. Our goal is to resolve the class conflict where necessary while sharing equivalent classes between

```
 1: procedure ClassRename
Input: A project set $P = \{p_1, p_2, \ldots, p_n\}$
Output: A renamed project set $P' = \{p'_1, p'_2, \ldots, p'_n\}$,
        where $\forall i \in \{1, \ldots, n\}. p_i = p'_i \wedge \textsc{Conflict}(p'_i, P') = \emptyset$
 2:     for $i \leftarrow 1, n-1$ do
 3:         $P \leftarrow P / p_i$
 4:         $worklist\ w \leftarrow \emptyset$
 5:         $queue\ q \leftarrow \emptyset$
 6:         $w \leftarrow \textsc{Conflict}(p_i, P)$
 7:         $q \leftarrow w$        ▷ add each element of w to q for renaming
 8:         while $w \neq \emptyset$ do
 9:             Pick and Remove $cl$ from $w$
10:             for all $cl' \in \textsc{depends}(cl, p_i)$ do
11:                 if $cl' \in \textsc{Shared}(p_i, P)$
12:                     & $cl' \notin q$ then
13:                     $q.enque(cl')$
14:                     $w \leftarrow w \cup$
         $\{cl'' \in \textsc{depends}(cl', p_i) | cl'' \notin q \wedge cl'' \in \textsc{shared}(p_i, P)\}$
15:                 end if
16:             end for
17:         end while
18:         $p'_i = \text{renameProject}(q, p_i)$
19:         ▷ make normal renaming substitution of $p_i$ for all classes in $q$
20:     end for
21: end procedure
```

Figure 3: Class Renaming Algorithm

projects. Fig. 2.(b) shows a centralization result without duplicating the code that can be shared.

D. Class renaming algorithm

Fig. 3 shows our renaming algorithm. The input of this algorithm is a set of projects to be centralized. The output is the renamed projects containing no conflict classes, and each of them is equivalent to the project before renaming. For a project set of size $n = |P|$, the algorithm iterates and renames each of the first $n - 1$ projects. The worklist w is used for traversing the class dependency relation. The queue q stores the classes to be renamed.

All the conflict classes of the each project are put into q for renaming. Their renaming effect then propagates to all the shared classes. If some class referenced by a shared class is renamed, the code of shared class changes, and it can no longer be shared. The renaming effect fully propagates until the worklist w becomes empty. After finding all the classes needing renaming, $\text{renameProject}(q, p_i)$ in Fig. 3 performs normal renaming substitution on project p_i according to the renaming queue.

The class renaming algorithm is guaranteed to terminate. Each class of a project p_i is added to worklist at most once. The output condition is also guaranteed to hold. There is no class conflict because all conflict classes and their propagation effect are resolved. In addition, each project before and after renaming is equivalent by normal substitution. For complexity, we consider an analysis of n projects, which includes m class names in total. In the worst case, each class name exists in n projects. If the calculation for conflicting and shared classes uses pairwise comparison, the algorithm costs $\mathcal{O}(m * n^2)$. After class renaming, no project holds a conflict class and all projects can be centralized by taking the union of all their classes.

IV. Shutdown semantics

Shutdown semantics [1] concern the termination of the centralized application. Invoking Java standard library methods like Runtime.exit and Runtime.halt [6] terminates the entire VM. In original program, each process runs on a different VM so that while some process invokes these methods to terminate, other processes may continue running. After centralization, all processes are wrapped as threads and run on one single VM. Some centralized process that invokes such shutdown methods terminates all the other processes and the entire VM without proper transformation. The shutdown behavior preservation of centralized program involves two issues: (1) If a process exits it only terminates all its threads. (2) All resources held by the process are released. The second issue is addressed in [1]. For the first issue, a simple way for a thread to terminate itself is to throw an exception of type *ThreadDeathException*. However, killing other threads in Java is difficult [6].

We adopt the interruption mechanism to kill a thread. This needs collaboration between the threads that send and receive the signal. Given two threads A and B, to kill B from A, A first calls $B.interrupt$ to send interruption signal to B. If B has been enabled to run, it can receive the signal from A and exit according to its status:
(1) When blocked on an interruptible invocation like wait() or sleep(), the signal triggers an *interruption exception* by the JVM. By catching this exception, B can exit safely.
(2) When blocked on uninterruptible actions like IO.read(), B has to be unblocked by closing the resource it is blocked on in order to check its interruption flag status to exit.
(3) When not in a blocking state, B can check its interruption flag to exit. If B is not enabled to run, it does not receive the interruption signal from A. The interruption caused by the centralizer and user should also be distinguished. These issues can be solved by adding additional flags to B.

To correctly kill a thread covering all these cases, we need to instrument the code of each thread (not necessarily between each statement) to check its interruption flag to exit. Note that each wrapped process either performs an internal operation in its local space that is unobservable by other processes, or it communicates with other processes. The internal operations cannot change the state of another process. Therefore, code instrumentation to check the interruption status is only needed before and after some key communication statements like ServerSocket.accept. If a thread calls shutdown methods to terminate, it sends the interruption signal to all other threads of the same centralized process. When the other threads are scheduled to run, they can check their interruption status to exit safely.

V. Implementation and experiments

This section presents the implementation and experimental results of our centralization tool.

Table I: EXPERIMENTAL RESULTS OF CENTRALIZATION

	#Classes	#Unique cl.	Shared Name		#Renamed	#Static Fields		#Static Sync Method
			#Same Code	#Diff Code		(#Transformed)		
Netx-0.4	91	12	37	42	57	109	69	0
Netx-0.5	88	9				135	101	0
Kryonet-2.08	79	8	12	59	67	20	4	0
Kryonet-2.20	104	33				23	4	0
Xnio-2.1.0CR1	74	7	21	46	66	46	46	0
Xnio-2.0.0CR2	72	5				46	46	0
Ganymed-ss2-build209	115	0	94	21	75	182	84	3
Ganymed-ss2-build210	133	18				257	45	3
Edtftpj-2.3.0	106	1	80	25	51	367	151	10
Edtftpj-2.4.0	113	8				240	94	10
Mime4j-core-0.7.1	61	0	60	1	26	118	59	1
Mime4j-core-0.7.2	61	0				235	117	1
Jsmpp-2.0	201	0	191	10	134	811	405	2
Jsmpp-2.1	202	1				407	204	2

A. Implementation

We implement a centralization tool by transforming Java bytecode based on the ASM bytecode library [4]. Before centralization starts, the centralizer parses a user-defined script into a Java startup class file, which defines how the each process starts. The centralizer transforms the classes of each project as described in the script, as defined in previous work [11], [1]. After transformation, the centralized program can be executed from the synthesized startup program.

The centralization tool is implemented into four passes. The first pass reads the class files to build internal data structures; the second pass implements project centralization by the using class renaming algorithm in Fig. 3. The third pass transforms static fields and class descriptors [11], [1]. We refine the centralization of static fields by transforming the final static fields that store mutable data. Final static fields storing immutable data do not cause data races. The fourth pass performs transformation to preserve startup and shutdown semantics. For the startup semantics, the main issue is to ensure components start up in the desired order such that dependencies between components are satisfied; for example, a server needs to be ready to accept connection before its clients are started. We limit our code instrumentation to a few key network functions. Whenever some component tries to connect to a port, it creates an external process to check whether the port is open. If the port is open, it continues to connect, otherwise it waits until the port is open. This approach does not modify Java network library and it scales up for larger network applications. For the shutdown semantics, we have manually verified various situations that a thread successfully terminates after receiving the interruption signal as described in Section IV. Process resource registration and release are also implemented as described in previous work [1].

B. Experiment

We apply our centralization tool on some existing Java network projects as benchmarks. The experiment centralizes two versions of the each project as a group. Table I shows the results. The changes in the number of classes in each group indicate that some classes are removed, or new classes are added. The number of *Unique* classes shows the details of such changes. Column *Shared Name* shows the project version update does not change many class names. Most class names remains the same; some classes modify their implementations. These numbers are listed in column *Same Code* and *Diff Code*, respectively. Column *Rename* displays the number of classes are renamed for each group by the renaming algorithm in Fig. 3. This algorithm renames all the conflict classes. Whether a shared class is renamed is decided by the class dependency. The experimental result shows that about half of the shared classes can still be shared after renaming. The last two columns show the number of static fields and static synchronized methods. The data indicates that manually searching and modifying these fields needs lots of effort even for two projects. Automatic tool support is therefore very useful for centralization.

C. Applications

We present two applications of the centralization tool based on our experiments in this section.

1) Centralization with JPF: To show our centralization tool performs a correct transformation, we first repeat previous experiments using centralization with Java PathFinder (JPF) [1]. These experiments were run on the Echo client/server, Daytime client/server, Alphabet client/server [2], Chat Server as test beds. We can correctly find the all the described bugs.

We proceed to seed some common faults into these benchmarks. One of these faults is a program crash caused by a truncated message. Consider the Echo client/server example: In the original protocol, the server first initializes itself and waits for the two clients to connect. When a client connects to the server, the server sends the same message back to the client. The client exits after receiving the echo message from server. The server terminates after it serves two clients. In the faulty version, we change the code of one client and server. One client is modified to crash if the messages it sends and received are not the same length, and the other client's code remains the same. On the server side, we inject a fault to send a truncated message back to the client with a low probability. A modular verification using JPF, analyzing either client side or server side separately as implemented by net-iocache [2], cannot detect such bugs. Previous centralization tools are not applicable as they do not support applications with a class conflict. After using our centralization on all network peers, we can successfully detect these bugs by JPF.

2) Centralization with profiling tools: Profiling is important for understanding the runtime behaviors of network applications. However, existing profiling tools like JRAT [7] only support a single process. Although profiling each process of a network application separately is possible, such analysis is difficult to automate and introduces overhead to start and destroy multiple JVMs. Profiling the centralized program avoids such overhead by running on single VM. The performance of each component and the execution traces of the whole network application can also be retrieved by existing profiling tools. We have performed some experiments to use the JRAT on centralized distributed applications. The result shows that centralization automates such integrated profiling of different components easily.

VI. Related work

Stoller [11] initially proposes to use centralization for verifying distributed Java applications. Artho et al. [1] improves the accuracy of centralization for such verification by JPF. However, the implementation uses the outdated SERP bytecode library [9], which makes it unable to work on current Java applications.

Compared with previous work, we intend to build an automatic centralization tool for a general-purpose analysis of distributed applications. As resolving class conflicts is essential for centralizing larger distributed applications, we propose our solution and implement it in our centralization tool. Our solution of startup and shutdown semantics does not depend on specific tools, either. Although the large state space of distributed applications limits software model checkers to small cases, our general centralization approach enables many existing dynamic analysis tools to analyze larger distributed applications.

Other work on verifying distributed applications includes net-iocache [2] and modeling the Java class loader [10], both of which target JPF. Compared with the centralization approach, net-iocache runs faster by sacrificing the completeness of verifying all execution traces. However, this limits net-iocache to not being able to find some bugs that centralization can.

Modeling multiple processes by using separate class loaders is proposed as a new feature in JPF v7 [10]. It models class loaders to separate process name spaces. Currently, JPF v7 is under development. We will compare it with our centralization approach after it is released.

VII. Conclusion and future work

In this paper, we advance centralization as a general analysis framework for distributed Java applications. We formalize and solve the class conflict to support centralization on applications containing multiple versions of a given class. We also propose a cleaner and complete solution for shutdown semantics. We implement an automatic centralization tool and validate it empirically. The experiments show that our tool works correctly, and support the usefulness of tool automation. The experiment using Java PathFinder shows that some defects can be detected by analyzing the centralized program but not without centralization.

Future work includes running experiments on various dynamic analysis tools, finishing the remaining implementation of the proposed shutdown semantics, and optimizing the class renaming algorithm.

Acknowledgment

This work was supported by Global COE and SEUT ⌈Secure-Life Electronics⌋ Program from MEXT, Japan. Thanks also go to Nastaran Shafiei for her comments.

References

[1] C. Artho and P.-L. Garoche, "Accurate Centralization for Applying Model Checking on Networked Applications," in *Proc. the 21st IEEE/ACM Int'l Conf. Autom. Softw. Eng.*, Washington, DC, USA, Sep. 2006, pp. 177–188.

[2] 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. the 2009 IEEE/ACM Int'l Conf. Autom. Softw. Eng.*, Washington, DC, USA, 2009, pp. 447–458.

[3] U. M. Borghoff and J. H. Schlichter, *Computer-Supported Cooperative Work: Introduction to Distributed Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2000.

[4] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A code manipulation tool to implement adaptable systems," in *Adaptable and extensible component systems*, Grenoble, France, Nov. 2002.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005.

[6] Java Platform, Standard Edition, V6 API Specification, "http://docs.oracle.com/javase/6/docs/api/."

[7] Java Runtime Analysis Toolkit, "http://jrat.sourceforge.net/."

[8] JCarder, "http://www.jcarder.org/."

[9] Serp, "http://serp.sourceforge.net/."

[10] N. Shafiei and P. Mehlitz, "Modeling class loaders in java pathfinder version 7," *SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, Nov. 2012.

[11] S. D. Stoller and Y. A. Liu, "Transformations for model checking distributed Java programs," in *Proc. the 8th Int'l SPIN workshop on Model checking of software*, New York, NY, USA, 2001, pp. 192–199.

[12] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, pp. 203–232, Apr. 2003.