reprinted from: IPSJ Transactions on Programming 6 (4) : 65–77 (2013)
IPSJ Online Transactions Vol.7 1–13 (Jan. 2014)
[DOI: 10.2197/ipsjtrans.7.1]

Regular Paper

Improving Automatic Centralization by Version Separation

Lei Ma[1,a) Cyrille Artho[2,b) Hiroyuki Sato[1,c)

Received: April 10, 2013, Accepted: September 2, 2013, Released: January 8, 2014

Abstract: With today's importance of distributed applications, their verification and analysis are still challenging. They involve large combinational states, interactive network communications between peers, and concurrency. Although there are some dynamic analysis tools for analyzing the runtime behavior of a single-process application, they do not provide methods to analyze distributed applications as a whole, 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 existing tools. In this paper, we improve the accuracy of centralization. Moreover, we extend it as a general framework for analyzing distributed applications with multiple versions. First, we formalize the version conflict problem and present a simple solution, and further propose an optimized solution to resolving class version conflicts during centralization. Our techniques enable sharing common code whenever possible while keeping the version space of each component application separate. Centralization issues like startup semantics and static field transformation are improved and discussed. We implement and apply our centralization tool to some network benchmarks. Experiments, where existing tools are used on the centralized application, prove the usefulness of our automatic centralization tool, showing that centralization enables these tools to analyze distributed applications with multiple versions.

Keywords: distributed application, dynamic analysis, software model checking, version conflict

1. Introduction

With today's importance of distributed applications, analyzing them is still 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 to be identical. Such nondeterminism from concurrency makes the run-time behavior of distributed application difficult to understand, predict, debug, and verify. This problem becomes more exacerbated if multiple threads inside a process are involved, creating concurrency inside a process as well as between processes. Considering 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 very important. Although many existing tools like Java PathFinder (JPF) [20], Java Interactive Profiler (JIP) [19] work on single-process applications, they do not support multi-process applications. If powerful analysis tools that support a single process were available to multiple processes, development and analysis of distributed systems would become easier.

Process centralization [1], [17] is a solution to enable existing tools to analyze multi-process applications without a version conflict. It transforms a multi-process application into a single process one with the equivalent runtime behavior. **Figure 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 separately serve each connected client. After centralization, all processes are wrapped as threads and run as one process. Centralization was initially proposed to verify distributed applications exhaustively. However, the large combinational states limit possible analysis to small applications. We propose using centralization for a general (not necessarily exhaustive) analysis of distributed applications.

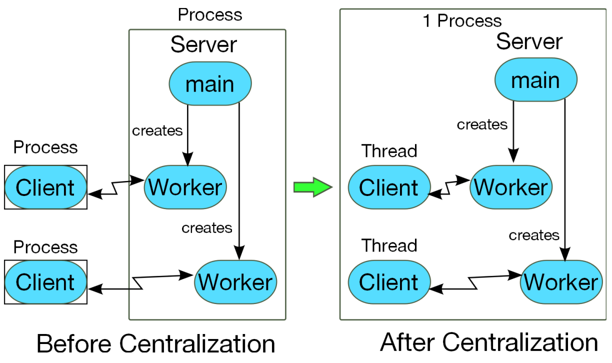


Fig. 1 Process centralization example.

Centralization enables many distributed applications to be available to existing tools and reduces the difficulty for analyzing them. For example, in a single-process debugger, a distributed application cannot be paused in a single step; when centralized this becomes possible. Other dynamic verification tools such as Java Race Detector [14] and JCarder [10] detect data races and deadlock bugs for single-process applications. However, they

[1] The University of Tokyo, Bunkyo, Tokyo 113–8658, Japan
[2] Research Institute for Secure Systems AIST, Amagasaki, Hyogo 661–0974, Japan
a) malei@satolab.itc.u-tokyo.ac.jp
b) c.artho@aist.go.jp
c) schuko@satolab.itc.u-tokyo.ac.jp

do not support multi-process applications. Meanwhile, profiling tools [8], [19] are useful for gathering the runtime performance of distributed applications. They only provide methods to separately analyze each component. This brings additional overhead by creating and destroying multiple VMs and lacks scalability. Because a centralized application runs on one single VM, these profiling tools can collect all the related profiles and scale to larger distributed applications. Finally, visualization tools [11], [18] are useful for understanding the runtime behavior of applications. They extract call graphs of distributed applications automatically, which helps to understand how its multiple components interact. Centralization makes it possible to visualize distributed systems also in this case.

There are currently no automatic centralization tools available. Previous centralization tools [1], [17] are outdated and unable to work on current Java applications. Previous works mainly use centralization to verify multi-process applications with JPF [20]. Certain aspects of the implementation such as system startup and shutdown are targeted to JPF and not able to work with other analysis tools [1]. When moving beyond JPF, larger systems can be supported, making tool automation all the more important.

Furthermore, one essential centralization issue, the problem of classes with multiple versions in different components, is not addressed by previous work. This commonly occurs in component-based systems, where different parts are developed independently and thus may use different versions of component classes. Lacking the support of such features essentially limits the application scope of centralization.

In this paper, we improve centralization as a framework for analyzing distributed applications with multiple versions. We formalize the class version conflict issue and address it with a simple algorithm and an optimized algorithm. Our technique shares common code whenever possible while keeping the version space of each application separate. In addition, centralization issues like process startup semantics and static fields are also refined and discussed. The solution is implemented in an automatic centralization tool. Experiments prove the effectiveness of tool automation, showing that the centralized application is more efficient in terms of run time and memory compared with the counterpart without centralization. Experiments with JPF demonstrate that out approach enables existing tools to verify a distributed application with multiple versions, showing that some defects can be found with centralization that are missed with single-process analysis.

The remainder of this paper is organized as follows. Section 2 summarizes centralization issues. Section 3 formalizes issues with multiple versions of a class, and explains our solution to resolve a version conflict in centralization. Section 4 explains the implementation of the centralization tool. Section 5 illustrates the experiments by using our tool. After discussing related work in Section 6, Section 7 concludes and discusses future work.

2. Centralization Issues

The term *distributed application* contains three aspects [3]: firstly, it means an application whose functionality is split into a set of cooperating, interacting functional units. Each unit runs as a process that has its internal state (data) and operations to manipulate the state. Secondly, these functional units can be assigned to different machines. A single machine, however, may host several functional units at the same time. Finally, the functional units communicate with each other through a network.

On modern operating systems, distributed applications are implemented as a system using multiple processes. They usually run on different hosts and communicate over a network. Process 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 communications between the transformed processes are internalized.

This paper is concerned with the centralization of programs written in Java [7], a 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 communications, although their implementation 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 resolved.

(1) *Version separation*. A component-based system consists of multiple components (including the application main component and library components), where each component is developed and managed independently. In software maintenance and evolution, each component of a component-based system needs to be continually changed over its lifetime to improve its functional capability to satisfy the users' requirements [13]. This may result in conflicts between different versions of the same product that are active at the same time. The problem becomes more exacerbated in a distributed system, where installations are duplicated over many peers. Each application is asynchronously updated in a "rolling update." This creates multiple versions of the components in a system, including both their used libraries and application code. Dumitraş et al. [4], [5] point out that most update failures are not caused by a software defect, but by version conflicts during the update procedure where the main code or library code changes.

Before centralization, each component application runs as a process (a functional unit in the distributed system) on its own Virtual Machine (VM) and locally holds its own version of each class. Because a centralized program runs on a single VM and each class is loaded and defined once, naive centralization may cause some processes to work incorrectly, if multiple versions of a class with the same name exist. We formalize the version conflict problem and address it by using two approaches: a simple solution and an optimized solution. They are explained in Section 3, with experimental evaluations in Section 5.

(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

single instance of a given class. Accessing these data by different processes without proper separation in the centralized program would cause data races. Therefore, centralization should keep the memory space of each process separate. Previous work addresses these issues; we discuss our refinement in Section 4.

(3) *Runtime behavior*: Startup and shutdown. Centralization wraps each process of an original program as a group of threads and starts them in the same way before the centralization. We denote each group of such threads by a *centralized process* which has the same runtime behavior as its corresponding process before centralization. Multiple centralized processes run on a single VM after centralization. To manage different centralized processes, Stoller [17] proposes a solution by defining the additional class *CentralizedProcess*. Each application in the original program is wrapped as a *CentralizedProcess* by centralization. This class holds a unique field as the *process ID*, which is used to identify each application at runtime.

The main issues are to start centralized processes in a required order and to preserve the shutdown semantics after centralization. For the analysis of network applications, ensuring that a server is initialized before clients try to connect is important. Otherwise, the client exits prematurely after failing to connect to the server. Previous work [1] solves this issues by modeling the network library. That solution is specific to JPF and difficult to generalize. We discuss our solution in Section 4.

Shutdown semantics [1] concern the termination of the centralized application. In the Java standard library, invoking methods like Runtime.exit and Runtime.halt [9] terminates the entire VM: the first one runs any previously registered *shutdown hooks*, and tasks that free resources during application shutdown; the second one halts the VM abruptly, without freeing any resources [7]. Each process in the original program runs on a different VM. If one process calls Java library methods Runtime.exit and Runtime.halt to terminate, other processes could continue running. After centralization, all processes are wrapped as threads and run on one single VM. A process that invokes a shutdown method terminates the entire VM and all the other processes. This changes the shutdown behavior of the original program. Centralization should preserve the shutdown behavior of the original program by proper transformation. This involves two issues:

(1) If a process exits, it only terminates its own threads.
(2) All resources held by the process are released and the shutdown hooks are executed if necessary.

The second issue is discussed by Ref. [1]. The resources for each centralized process need to be registered at a few key functions that allocate resources. Similarly, a shutdown hook should also be registered as a wrapped thread for each centralized process through code instrumentation. Whenever a centralized process calls Runtime.exit to exit, it invokes all the registered shutdown hooks and closes all the resources of the process that are not closed. We discuss the first issue in Section 4.

3. Version Separation

The usage of slightly different versions of components is common in component-based systems. Centralization is incorrect without properly separating the class namespace for each component application. In this section, we first formalize the version separation problem and propose a simple algorithm to solve this issue. Furthermore, we propose an optimized algorithm that is more effective in sharing common code and saving storage.

3.1 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 $cl_1.name$ is identical to $cl_2.name$ and $cl_1.code$ is identical $cl_2.code$.

Definition 1 A *project* is a set of classes. Given a project p, we write $\#p$ as its cardinality, and denote a class cl in a project p by $p.cl$. No two classes in a project have the same name.

A project presents an abstract view of the class repository of a component. The code repository of a component application is composed by multiple components, each of which can be represented by a project. Furthermore, the combination of all components can be represented by one *centralized* project by merging small projects. This paper considers the code repository of each component application in a distributed system as one project. Two component applications may use the code from either the same project or different projects.

Definition 2 Let p be a project. We define $\mathrm{NAME}(p) = \{cl.name | cl \in p\}$ as the set of all class names in p. For a class name $cln \in \mathrm{NAME}(p)$, we define $\mathrm{GetClass}(p, cln) = p.cl$, where $p.cl.name = cln$, as a function get the class named cln in the project p. Similarly, let P be a set of projects. We define $\mathrm{NAMES}(P) = \cup_{p \in P}\mathrm{NAME}(p)$ as the set of all the class names in P, and $P \uparrow cln = \{p \in P | cln \in \mathrm{NAME}(p)\}$ as the set of all those projects that contain the class named cln.

Definition 3 Let p be a project, and cln_1 and cln_2 be two class names. Project renaming substitution $p[cln_1/cln_2]$ is defined as a project in which cln_1 in p is substituted for cln_2, including both class names and their references in the code. A renaming substitution $p[cln_1/cln_2]$ for p is a *normal substitution* iff $cln_1 \notin \mathrm{NAME}(p)$ and $cln_2 \in \mathrm{NAME}(p)$.

Definition 4 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.
It is not difficult to prove that it is reflexive, symmetric, and transitive.

Definition 5 Project *centralization* of a project set P transforms P into one single project p_{centra} such that $\forall p \in P. \exists p' \subseteq p_{centra}. p = p'$. We denote all the centralized results of P that satisfy such a condition by $\mathrm{CENTRA}(P)$.

Project centralization requires preservation of class name and version space for each project. Each project consists of several small projects, each of which represents the code repository of a component in the component application. Each component application that runs on the its original project can also run on the centralized project with the same runtime behavior. The projects (representing the code repositories of multiple component applications) to be centralized can either be the same or different in their internal components.

The term *process centralization* is defined as follows.

Definition 6 *Process centralization* is the transformation of multiple processes into a single one with the equivalent runtime behavior.

Different from project centralization, process centralization [1], [17] simulates runtime behavior of multiple component applications by a single application with equivalent runtime behavior by program transformation. They assume that all the applications run under the same project, where each class has only one version and no version conflict occurs.

On the other hand, project centralization considers sharing the common code repository of component applications to save storage while keeping their version space of each application separate so that no version conflict occurs. We propose using project centralization to enable process centralization to analyze component applications with multiple versions of the same component.

Definition 7 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 $\mathrm{DEPENDS}(cl, p) = \{cl' \in p | cl \rightarrow cl'\}$ to be the set of all classes in p that depend on cl.

The class dependency represents the class reference relation in a project. Given two classes cl_1 and cl_2 with the relation $cl_1 \rightarrow cl_2$, if cl_1 is renamed, all its references in cl_2 must also be renamed to preserve the dependencies.

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

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

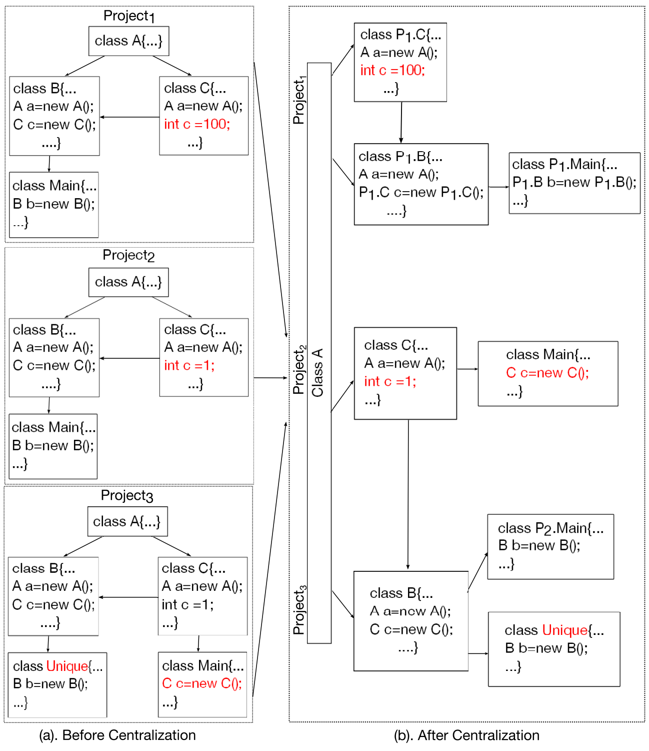
(2) *Conflict Class*. $\mathrm{CONFLICT}(p, P) = \{cl \in p | \exists q \in P. (cl.name \in (\mathrm{NAME}(p) \cap \mathrm{NAME}(q)) \wedge p.cl \neq \mathrm{GetClass}(q, cl.name)\}$. The name of a conflict class appears in multiple projects including p, but with different implementations.

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

3.2 Example

Figure 2 (a) shows an example for centralizing three projects. The edges in a project represent the class dependencies. In a project, we draw a directed edge from class cl_1 to cl_2 if there exists a dependency relation $cl_1 \rightarrow cl_2$. Project$_1$ and Project$_2$ share most of the classes except different versions of class C are used. 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 classes in all projects. The cases of classes C and *Main* are more complex: the class C is a conflict class in Project$_1$, but it is both a shared and conflict class in Project$_2$ and Project$_3$. Similarly, class *Main* is a conflict class in Project$_3$, and it is both a shared and conflict in Project$_1$ and Project$_2$.


Project1
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(); ...}
Project2
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(); ...}
Project3
class A{...}
class B{... 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(); ...}
Project1
Project2
Class A
Project3
class P1.C{... A a=new A(); int c =100; ...}
class P1.B{... A a=new A(); P1.C c=new P1.C(); ....}
class P1.Main{... P1.B b=new P1.B(); ...}
class C{... A a=new A(); int c =1; ...}
class Main{... C c=new C(); ...}
class P2.Main{... B b=new B(); ...}
class B{... A a=new A(); C c=new C(); ....}
class Unique{... B b=new B(); ...}
(a). Before Centralization
(b). After Centralization
Fig. 2 Project centralization example.

3.3 Project Centralization and Class Renaming

Consider a general scenario of centralizing processes using the project set $P = \{p_1, p_2, \ldots, p_n\}$, where one or more processes start from each project. Direct centralization of these processes is incorrect if $\exists p_i \in P.\mathrm{CONFLICT}(p_i, P) \neq \emptyset$. We therefore propose to make project centralization before process centralization. After project centralization of P, all its projects are represented by one single project, where each class has only one version and no version conflicts exist for process centralization. The main issue of project centralization is to properly separate class version and name space for each project. To scale up to practical applications, we adopt the class renaming approach.

A trivial solution would entail renaming all classes, duplicating all code for each project. However, code duplication would cost more storage to represent the code repository and consumes large runtime memory by loading more classes into Java VM. This causes such an approach difficult to scale up to larger applications. 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, saving both storage and runtime memory. Therefore, it is beneficial to share the common class codes. Our goal is to resolve the class version conflict where necessary while sharing equivalent classes among projects. Figure 2 (b) shows a project centralization result without duplicating the code that can be shared. The trivial solution produces 13 classes. However, it is only necessary to keep one version of class A after project centralization. Similarly, we can keep two versions of class B and C. One version is shared by Project$_2$ and Project$_3$, and the other version is used for Project$_1$. In Fig. 2(b), we need only 9 classes.

3.4 The Simple Project Centralization Algorithm

The main issue of resolving version conflict is to properly separate the class version for each project. This entails renaming the conflict classes and all their references to separate their versions. However, such renaming may cause shared classes not shareable anymore, as their internal references to other classes are renamed differently across projects. Consider the example in Fig. 2: Project$_1$ and Project$_2$ can share class B before project centralization. They have to rename their class C to a different name to solve version conflict, though. After that step, B cannot be shared anymore as it references C. Therefore, it is necessary to rename the conflict classes and propagate their renaming effect in each project.

In this section, we propose a simple project centralization algorithm as shown in **Fig. 3**. 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. Given a project set P with $\#P = n$, the algorithm iterates and renames each of the first $(n-1)$ projects. We use the worklist w for traversing the class dependency relation, and the queue q for storing the classes needing renaming, respectively.

For each project, the algorithm first calculates all the conflict classes of the current project and put them into q for renaming. For each conflict class, its renaming effect then propagates to all the shared classes. The renaming effect fully propagates until the worklist w becomes empty. After finding all the classes needing renaming, renameProject(q, p_i) in Fig. 3 performs normal renaming substitution on project p_i according to the renaming queue q.

This algorithm is guaranteed to terminate. Each class of a project p_i is added to the worklist at most once and only those classes that are either shared or conflicting can be added to the worklist. The output condition is also guaranteed to hold. There is no class version conflict because all conflict classes and their propagation effect are resolved. In addition, projects before and after renaming are equivalent by normal substitution.

```
 1: procedure SimpleProjectCentralization
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 $\forall i \in \{1, \ldots, n\}. p_i = p'_i \wedge \text{Conflict}(p'_i, P') = \emptyset$
 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:       $w \leftarrow \text{Conflict}(p_i, P)$
 7:                                  ▹ add the conflict classes of $p_i$ into worklist
 8:       $q \leftarrow w$                      ▹ add each element of $w$ to $q$ for renaming
 9:       while $w \neq \emptyset$ do
10:           Pick and Remove $cl$ from $w$
11:           for all $cl' \in \text{depends}(cl, p_i)$ do
12:               if $cl' \in \text{Shared}(p_i, P)$
13:                   $\wedge\ cl' \notin q$ then
14:                   $q.enque(cl')$
15:                   $w \leftarrow w \cup \{cl'\}$
16:               end if
17:           end for
18:       end while
19:       $p'_i$ = renameProject$(q, p_i)$
20:            ▹ make normal renaming substitution of $p_i$ for all classes in $q$
21:   end for
22:   $P' \leftarrow \cup_{i=1}^{n} p'_i$
23: end procedure
```

Fig. 3 Simple project centralization algorithm.

For complexity, we consider analyzing a project set P with $\#P = n$ which includes m class names in total. All input projects are internal data structures that represent class raw files. The class classifications and dependency relations are pre-calculated during the preprocessing phase. The complexity for checking the existence of a class in set Conflict(p, P) or Shared(p, P) is $O(m)$. The dependency relation set DEPENEDS(cl, p_i) for class cl in project p_i contains at most $(m-1)$ classes (excluding the class self dependency). In the worst case, the complexity for traversing the class dependency relation in the loop of worklist is $O(m^2)$. Therefore, the complexity for calculating the renaming decision of project set P is $O(m^2 \cdot n)$. After class renaming, no two projects hold conflict classes and all projects can be centralized into one project by taking the union of all their classes.

This algorithm correctly separates the version space of all input projects. However, it does not always output a satisfactory solution. The optimized solution separates the project version space while sharing classes whenever possible. A class in a project can be both a conflict class and a shared class. This algorithm does not distinguish a conflict class and a class that is both shared and conflicted. It simply renames the class as long as it is a conflict class.

Consider the example in **Fig. 4** (a). P is a project set to be centralized, where $P = \{p_1, p_2, p_4, p_3\}$ and each project $p_i \in P$ has one class A. There exist two versions of A in P, where p_1, p_2 hold one version, and p_3, p_4 hold the other version. We represent different versions of a class by different colors. The simple algorithm renames all A's in p_1, p_2, p_3, but not in p_4, resulting three classes after project centralization as shown in Fig. 4 (b). However, the optimized solution produces only two classes (the two versions of A): one is shared by p_1 and p_2 and the other is shared by p_3 and p_4 as shown in Fig. 4 (c).

The limitation of this algorithm is caused by a lack of version linkage of classes among all projects. Suppose there exists a class A in $P' = \{p'_1, p'_2, \ldots, p'_n\}$ with n versions. Theoretically, renaming any $(n-1)$ versions of A can resolve version conflict. However, properly selecting the $(n-1)$ versions for renaming is a difficult problem. Whenever a class is renamed, its effects propagate in the project, which may result in more classes that must be renamed. To approach an optimal solution, we need to search all possible version combinations of classes in all projects for renaming. When centralizing a project with m names each with n

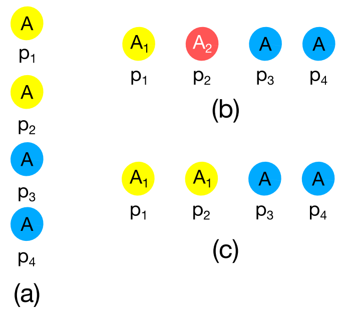


Fig. 4 Simple algorithm example.

versions, the complexity for searching the optimal combination of class renaming actions is $O(n^m)$. Algorithm in Fig. 3 approximates this by simply renaming all classes that are both shared and conflicting.

3.5 The Optimized Project Centralization Algorithm

To obtain an optimized solution, we adopt a *P-graph* to represent the project set to be centralized. Compared to the simple project centralization algorithm that propagates the renaming effect by using the classification of a class, the graph based algorithm propagates version constraint by using *set partition* and *conflict edges*.

Definition 8 A *partition* of a project set P is a set of disjoint project sets $T = \{t_1, t_2, \ldots, t_k\}$ satisfying:
(1) $\forall t_i, t_j \in T. i \neq j \Rightarrow t_i \cap t_j = \emptyset$.
(2) $\bigcup_{i=1}^{k} t_i = P$.
(3) $\forall t \in T. t \neq \emptyset$.

Let T be a partition of project set P. We write $\overline{T}$ as the project set by ignoring its partition. That is $\overline{T} = P$.

Definition 9 Let $T = \{t_1, t_2, \ldots, t_i\}$ and $Q = \{q_1, q_2, \ldots, q_j\}$ be two partitions of a project set P. T is *finer-equal* than Q denoted by $Q \preceq_p T$ iff $\forall q \in Q. \exists S \subseteq T. \overline{S} = q$. We define the least upper bound of T and Q as $T \sqcup_p Q = \{t \cap q | t \in T \wedge q \in Q \wedge t \cap q \neq \emptyset\}$, which is also a partition of P.

Definition 10 Let P be a project set. A *partition structure* PS in P consists its name, denoted by $PS.name$ where $PS.name \in \text{NAMES}(P)$, and a partition of $P \uparrow PS.name$, denoted by $PS.partition$. We write $\langle PS.name, PS.partition \rangle$ for the partition structure.

Definition 11 Let PS_1 and PS_2 be two partition structures. We define partial order $PS_1 \preceq_{ps} PS_2$ iff $PS_1.name = PS_2.name$ and $PS_1.partition \preceq_p PS_2.partition$. We define the least upper bound of PS_1 and PS_2, where $PS_1.name = PS_2.name$, as $PS_1 \sqcup_{ps} PS_2 = \langle PS_1.name, PS_1.partition \sqcup_p PS_2.partition \rangle$.

We define the *P-graph* of a project set P which uses the project set partitioning to represent the version constraint relations.

Definition 12 A *P-graph* $\langle N, E \rangle$ of a project set P is consisted of a node set N of partition structures, and an edge set E with each edge $e = (l, m) \in E$ (an edge from node l to m) associated with a project set denoted by $e.pset = \{p \in (P \uparrow l.name \cap P \uparrow m.name) | \text{GetClass}(p, l.name) \rightarrow \text{GetClass}(p, m.name)\}$ such that:
(1) $\text{NAMES}(P) = \{n.name | n \in N\}$.
(2) $\exists e \in E. e = (m, n)$ where $m, n \in N$ iff $e.pset \neq \emptyset$.

Let $G = \langle N, E \rangle$ be a P-graph of project set P. Each node $n \in N$ is a partition structure $\langle n.name, n.partition \rangle$ that represents all the versions of the class named $n.name$ in P. Its partition $n.partition$ keeps the version relation of these classes so that if two projects are in the different sets in $n.partition$, they hold a different version of the class named $n.name$. We denote all *successors* of a node n in G by $\text{Succ}(n) = \{m | (n, m) \in E\}$. For two nodes $m, n \in N$, the existence of an edge $e = (m, n)$ from m to n entails that the classes named $m.name$ and $n.name$ have a dependency relation in some project $p \in P$, and p occurs in both $P \uparrow m.name$ and $P \uparrow n.name$. Note that (m, n) and (n, m) represent different edges in E.

Figure 5 (a) shows the corresponding initialized P-graph to the

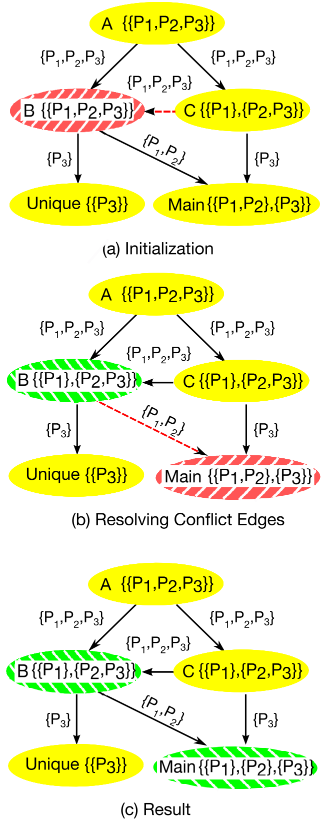


Fig. 5 Optimized algorithm example.

project set in Fig. 2 (a). The larger node is the partition structure node, inside which its name and partition are shown. For example, the node named A with its partition indicates that its name A exists in three projects P_1, P_2 and P_3. They all occur in the same set of partition, meaning all these projects have the same version of class named A. The label of an edge e in a P-graph shows its $e.pset$. For example, the edge from node B to $Main$ indicates that classes named B and $Main$ have a dependency relation in both P_1 and P_2, but not in P_3.

Correct project centralization requires keeping the version spaces of each project separate by renaming. Renaming a class also entails renaming all references to it accordingly. We define conflict edges to capture the effect that different versions of a class are not separated due to the version separation of another class that this class depends on.

Definition 13 Let $G = \langle N, E \rangle$ be a P-graph of P and $e \in E$ be an edge where $e = (m, n)$ and $m, n \in N$. The edge e is a *conflict edge* if $\exists p, p' \in e.pset. p \not\equiv p' \Rightarrow ((\exists t_i, t_j \in m.partition. i \neq j \Rightarrow p \in t_i \wedge p' \in t_j) \wedge (\exists t_k \in n.partition. p \in t_k \wedge p' \in t_k))$. We write IsConflictEdge(e) as a function to check whether e is a conflict edge in G. It returns *true* if e is a conflict edge, otherwise it returns *false*.

An edge $e = (m, n)$ in a P-graph of P is a conflict edge, if there exist two projects p and p' in $e.pset$ such that they occur in a different set of $m.partition$ but the same set of $n.partition$. A conflict edge captures cases where classes must be renamed to separate the version space of each project. For example, the dashed edges in Fig. 5 (a) is a conflict edge, where P_1 and P_2 occur in different sets in the partition of node C, but in the same set in the partition of node B.

A conflict edge $e = (m, n)$ can be resolved by refining *n.partition* into a finer partition such that e becomes a non-conflict edge. After a conflict edge is resolved, it may introduce additional conflict edges. To correctly separate the version space of each project, we need to ensure no unresolved conflict edges remain.

We propose an optimized project centralization algorithm by resolving conflict edges as shown in **Fig. 6**. The main procedure OptimizedProjectCentralization first initializes a P-graph by collecting all the class names, creating nodes and edges from an input project set (line 3–18).

```
 1: procedure OptimizedProjectCentralization
Input: A project set $P = \{p_1, p_2, \ldots, p_n\}$
Output: The centralized project $p_{centra}$, where $\forall p \in P.\exists p' \subseteq p_{centra}.p = p'$
 2:     $p_{centra} \leftarrow \emptyset$
 3:     $PGraph \leftarrow \emptyset$
 4:     $nameSet \leftarrow$ collectName($P$)                ▹ Collect all class names
 5:     $PGraph.nodeSet \leftarrow \emptyset$
 6:     for all $name \in nameSet$ do                  ▹ Build a node for each name
 7:         $PGraph.nodeSet \leftarrow PGraph.nodeSet$
                                  $\cup$ {createNode($name, P$)}
 8:     end for
 9:     for all $src \in PGraph.nodeSet$ do                        ▹ Add edges
10:         for all $targ \in PGraph.nodeSet/src$ do
11:             $tempSet \leftarrow \{p|p \in (P\uparrow src.name \cap P\uparrow targ.name)$
12:            $\wedge$GetClass($p, src.name$) $\rightarrow$ GetClass($p, targ.name$)}
13:             if $tempSet \neq \emptyset$ then
14:                 $(src, targ).set = tempSet$
15:                 $Pgraph.edgeSet \leftarrow Pgraph.edgeSet \cup \{(src, targ)\}$
16:             end if
17:         end for
18:     end for
19:     $components \leftarrow$connectionCalculation($PGraph$)
                        ▹ Calculate all connected components of $PGraph$
20:     for all $component \in components$ do
21:         resolveConflict($component$)
                  ▹ Resolve version conflicts for each connected component
22:         for all $node \in component$ do
23:             $p_{centra} \leftarrow p_{centra} \cup$ renameByPartition($node$)
             ▹ All projects in the same set in $node.partition$ share one version
24:         end for
25:     end for
26: end procedure
27:
28: function resolveConflict($P$-$graph\ graph$)
29:     $SCCs \leftarrow$ calculateSCC($graph$)
                        ▹ Calculate the strongly connected components
30:     $TopoSCCs \leftarrow$ calculateTopologicalOrder($SCCs$)
31:     for $SCC \in TopoSCCs$ do
                                    ▹ Visit each $SCC$ in topological order
32:         resolveIncomingConflictEdge($SCC, TopoSCCs$)
                ▹ Resolve the incoming conflict edges from other components
33:         $worklist\ w \leftarrow$ getAllInternalConflictEdges($SCC$)
                        ▹ add all internal conflict edges of $SCC$ into $w$
34:         while $w \neq \emptyset$ do  ▹ Iteratively resolve all conflict edges inside the
    component
35:             Pick and Remove $edge$ from $w$, where $edge = (m, n)$
36:             if isConflictEdge($edge$) then
37:                 $n.partition \leftarrow n.partition \sqcup_p (m.partition\uparrow edge.pset)$
              ▹ Resolving a conflict edge by refining $n.partition$ according to
                  $m.partition$ and $edge.pset$
38:                 for all $l \in$ Succ($n$) $\wedge\ l \in SCC$ do
39:                     if isConflictEdge($(n, l)$) then
40:                         $w \leftarrow w \cup \{(n, l)\}$
41:                     end if
42:                 end for
43:             end if
44:         end while
45:     end for
46: end function
```

Fig. 6 Optimized project centralization algorithm.

To reduce the steps of resolving conflict edges, the algorithm calculates the connected components of the whole graph, and resolves each component separately. The conflict edges of each connected component are resolved by function ResolveConflict. To reduce the calculations of dependency cycles in the connected components of the P-graph, we compute the *Strongly Connected Component* (SCC) of the graph and resolve them in a *topological order* so that each SCC is only handled once. For each SCC, we first resolve the conflict edges from other components to the current SCC. Then, we resolve internal conflict edges iteratively by refining the partitions of nodes (lines 31–45) until no conflict edges exist.

The initialized P-graph of the project set in Fig. 2 (a) is shown in Fig. 5 (a). Refining the partition of B resolves the conflict edge from C to B, and it produces another conflict edge from B to *Main* as shown in Fig. 5 (b). The final result of this algorithm is shown in Fig. 5 (c). For classes A and *Unique*, it outputs one version. It outputs two versions of classes B and C, and three versions of class *Main*, which is also the optimal solution.

This algorithm is guaranteed to terminate. Consider a P-graph $G = \langle N, E \rangle$ built from project set $P = \{p_1, p_2, \ldots, p_n\}$ with its node set $N = \{n_1, n_2, \ldots, n_m\}$. We denote the total number sets in all partitions of N by $SET^\sharp = \sum_{i=1}^{m} \#n_i.partition$. After resolving each conflict edge of G, $SET^\sharp$ increases and $SET^\sharp$ reaches its maximal value when no conflict edge exists. At each iteration of the algorithm, it either terminates or resolves some conflict edges, which increases $SET^\sharp$. The maximum of $SET^\sharp$ is the finite value $\sum_{i=1}^{n} \#p_i$. Therefore, the algorithm is guaranteed to terminate in at most $(\#E \cdot n)$ iterations. Each of these steps to resolve a conflict edge costs $O(n^2)$. To analyze the complexity of resolving the initialized P-graph $G = \langle N, E \rangle$ of P, we need to combine the complexity of calculating connected components, strongly connected components, topological sorting of G, and the complexity of resolving all conflict edges. Therefore, the complexity is $O(\#E \cdot n^3 + \#N)$ in total.

4. Implementation

We implement our project centralization solution as a four-pass transformation tool. The centralization tool transforms the Java bytecode by using the ASM bytecode library [12]. Before centralization starts, the centralizer parses a user-defined script into a Java startup class file which defines how each process starts. The centralizer transforms all the classes of all projects as described in the script, as defined in previous work [1], [17]. After transformation, the centralized program can be executed from the synthesized startup program.

4.1 Class Statistics

The first pass reads in the classes from all projects and builds their internal data structures accordingly. Some statistical information like the number of class files, the size of each project, and the number of static fields, is calculated in this pass. This provides the user with information about the number of modifications during transformation.

4.2 Project Centralization

The second pass implements project centralization. It reads the data structure built in the first pass and performs project centralization. It provides options to use either the simple algorithm in Fig. 4 or the optimized algorithm in Fig. 6. After project centralization, all projects are represented by one single centralized project data structure for further transformation.

Currently, this phase does not support transforming dynamically computed class names used by reflection. Reflection [7] is widely used to dynamically load Java classes. For example, Java standard library methods like Class.forName(String classname) and ClassLoader.loadClass(String classname) load a class with a computed name at runtime. To support dynamic class loading by reflection, a project centralization tool needs to keep a renaming map (renaming decision) for each project to be centralized. Our tool implementation to support reflection is left as a future work.

4.3 Static Fields and Class Descriptors

The third pass transforms static fields and class descriptors as Refs. [1], [17]. For static fields, we transform them into arrays and add one extra dimension if the field is an array. We refine the initialization semantics of static fields by analyzing and transforming the static initializer. We also do not transform *static final* fields if they store the immutable data. The *static final* fields that store mutable data are transformed because they can still cause data races when multiple threads access such data. We also transform static fields that are generated by the Java compiler (synthetic fields). Previous work transforms all static fields without such distinctions.

For class descriptors used as locks, they cannot simply be duplicated like static fields [9]. We adopt the proxy lock approach [1]. Whenever a class descriptor is used as a lock, we use the proxy lock instead. The usage of a class descriptor as a lock or for reflection is distinguished by analyzing instructions of the bytecode. If the class descriptor is on the top of current stack frame and the next instruction is *monitorenter*, it uses the class descriptor as a lock to enter the critical region. Otherwise, the class descriptor is used for reflection which should not be transformed.

4.4 Startup and Shutdown Semantics

The last pass implements the startup and shutdown semantics. For the startup semantics, the main issue is to ensure the centralized processes start up in the desired order such that dependencies between them are satisfied; for example, a server needs to be ready to accept connection before its clients are started. A previous implementation [1] is specialized for Java PathFinder by modeling the Java network library. It does not work for other tools than Java PathFinder. Our solution limits the code instrumentation to a few key network functions. Whenever a component application tries to connect to a port, it creates an external process to check the port status. If the port is open, it continues to connect, otherwise it waits until the port is open. This approach does not modify the Java network library and can be applied to tools other than Java PathFinder.

Shutdown semantics require that a process that calls Java library methods Runtime.exit and Runtime.halt to terminate, only terminates all its threads while other processes may continue running. This requires killing all its threads belonging to the centralized process. In Java, 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 [9]. The interruption mechanism [6] is a clean way to kill a thread in Java, which needs the collaboration between the thread that sends the signal and the one that receives the signal. Currently, our implementation along this approach is in progress, and tool automation support is left as future work.

5. Experiments

To evaluate the effectiveness of our proposed centralization approach, we perform three experiments on existing Java network benchmarks by using our centralization tool. This section explains the experimental results. All experiments were run on an Intel Core i7 Mac 2.4 GHz with 8 GB of RAM, running MAC OSX 10.8.3 and Oracle's Java VM, version 1.7.0_21.

5.1 Comparisons of Centralization Transformation

To compare the transformation performance of the two proposed algorithm, we apply our centralization tool with their corresponding algorithm options to several existing Java benchmarks. To measure the effectiveness of a project centralization algorithm in sharing common classes, we define the *Sharing Factor* (*S.F.*) as the ratio of shared classes to output classes: $S.F. = \#ClassShared/\#OutputClass$. The larger of its value, the more classes are shared by the renaming algorithm. When $S.F = 0$, no class is shared by the algorithm; when $S.F = 1$, all classes of each project are shared.

The experiment first centralizes each project with one instance for each version. **Table 1** summarizes some statistical results, and also shows the transformation time and memory consumption of our tool. Column *Bytec. Size* lists the size of each benchmark in bytecodes. The data from column 3 to 6 shows that the class version conflict is a common problem in all benchmarks. Our centralization supports the transformation of projects with class version conflicts. The class number for each benchmark in column *Cl.*(*.class files) indicates that some classes are removed, or new classes are added during version changes. The number of unique classes in column *Uni.* shows the details of such changes. Column *Shared Name* shows that the project version update does not change many class names. Most class names remain the same; some classes modify their implementations. These numbers are listed in columns *Sa.* and *Dif.*, respectively. Column *Re.* lists the number of classes that are renamed for each group by our renaming algorithm. When centralizing two projects, there does not exist a class that is both shared and conflict. Therefore, both proposed algorithms produce the same solution. For each benchmark in this experiment, $\#ClassShared = \#Shared\,Name - \#Re.$ and $\#Output\ Class = \Sigma\#Classes - \#ClassShared$. The trivial renaming approach renames all classes and shares no classes, its *S.F* is therefore 0. The experimental results show that both proposed algorithms are more effective in sharing common code, with a value of *S.F* between 0.01 and 0.97.

Table 1 Experimental results of centralization.

Proj. versions	Bytec. Size [KB]	#Cl.	#Uni.	Shared Name		#Re.	S.F.	#Static Field (#Trans.)		#Static Sync Method	Simple Centra.		Optimized Centra.	
				#Sa.	#Dif.						time [s]	mem. [MB]	time [s]	mem. [MB]
Edtftpj-2.3.0	352.34	106	1	80	25	49	0.34	223	88	10	1.26	96.3	1.55	101.6
Edtftpj-2.4.0	390.49	113	8					240	94	10				
Ganymed-ss2-build209	304.88	115	0	94	21	42	0.42	124	44	3	1.15	92.6	1.37	93.0
Ganymed-ss2-build210	345.22	133	18					257	45	3				
Jsmpp-2.0	456.93	201	0	191	10	134	0.78	407	204	2	1.20	104.2	1.74	115.7
Jsmpp-2.1	458.35	202	1					407	204	2				
Kryonet-2.08	205.46	79	8	12	59	67	0.02	20	4	0	1.03	92.9	1.19	87.5
Kryonet-2.20	25.157	104	33					23	4	0				
Mime4j-core-0.7.1	153.64	61	0	60	1	1	0.97	118	59	1	0.63	77.4	0.75	86.4
Mime4j-core-0.7.2	153.63	61	0					115	59	1				
Xnio-2.0.0CR2	248.92	72	5	21	46	65	0.01	46	46	0	0.91	93.2	1.10	89.6
Xnio-2.1.0CR1	253.94	74	7					46	46	0				
Netx-0.4	240.35	91	12	37	43	53	0.17	109	69	0	0.91	81.7	1.19	97.1
Netx-0.5	246.13	88	9					110	179	0				

Columns *Static Field* and *Static Synchronized Method* show the number of static fields and static synchronized methods. This indicates that manually searching and modifying these fields need lots of effort even for two projects. Automatic tool support for centralization is therefore very useful.

Columns *Simple Centra.* and *Optimized Centra.* present the time and memory consumption of code transformation when adopting the corresponding algorithm in the tool. We run each experiment setting 50 times and take the average of result. In this experiment, we found that the simple algorithm runs faster, and consumes less runtime memory in most benchmarks.

To further compare the effectiveness of the two proposed algorithm of sharing common code and transformation cost, we run the experiment to centralize projects of each benchmark with a different number of instances per version to observe the *S.F.* value, transformation time, memory consumption, and the storage to save the centralized results. Each experiment is repeated 50 times and the averaged results are summarized in **Table 2**. To interpret the data, we take the project centralization of Edtftpj-2.3.0 and Edtftpj-2.4.0 as an example. We have four settings, from centralizing one Edtftpj-2.3.0 and two Edtftpj-2.4.0 projects, to seven instances of both Edtftpj-2.3.0 and Edtftpj-2.4.0. The tool transformation performance of the two algorithm options is listed and can be compared both vertically and horizontally. For the simple algorithm, as the number of projects increases, its *S.F.* value decreases, which indicates that some classes are not shared as the number of projects increases. The storage cost after project centralization increases, showing that it needs more storage to save centralized result. The concrete project sizes after transformation are shown in column 8 and 12, respectively. The optimized algorithm is more effective in sharing common code (larger *S.F.* value) and saving storage (less storage cost) than the simple algorithm in each setting.

As the number of projects for centralization increases, the transformation time and memory consumption of both approaches increase as listed in columns *Time* and *Mem.*. For the transformation time, the simple algorithm is faster for centralizing small number of projects such as the first setting with one instance of first version, and two instances of second version. Compared with the simple algorithm, the optimized algorithm takes more time to initialize the internal data structure and compute the renaming decision. As the number of projects for centralization increases, the simple renaming algorithm produces more classes to output, and it takes more time to store these files to disks. The optimized algorithm takes the advantage of sharing common code (outputting less classes) and runs faster to save the centralized results. By making similar analysis, we can draw the same conclusion for other benchmarks in Table 2. Therefore, the optimized algorithm based tool is more effective in sharing common code and more efficient to store the centralized results.

5.2 Comparisons of Runtime Performance

The second experiment applies our tool on actual networked applications to compare the runtime performance (including execution time and memory consumption) of the centralized applications transformed by two proposed algorithms and the corresponding one without centralization.

The networked applications used in this experiment are summarized as follows :

- The Echo server sends all input back to client. The Echo client is a test client that connects to the server and sends predefined text to it (RFC 862).
- The Daytime Sever returns the current time back. The Daytime client requests the current time from the server (RFC 867).
- The Chat server returns the input of one client to all connected clients. The chat client is a test client that connects to the server, sends predefined text to it, and disconnects after having received a certain number of lines.
- The alphabet server returns the nth letter of alphabet, and the client sends fixed requests.

All experiments of each setting are repeated 50 times (including both centralization transformation and execution of the centralized application), and the averaged results are summarized in **Table 3**. The column *Bytec. Size* lists project size of each component application in a benchmark in bytecodes. Consider the Echo

Table 2 Comparison of centralization transformation by two algorithms.

Proj. versions		#N.		Simple				Optimized			
				S.F. [%]	Time [s]	Mem. [MB]	Storage [KB]	S.F. [%]	Time [s]	Mem. [MB]	Storage [KB]
Edtftpj-2.3.0	2.4.0	1	2	69.3	1.43	106.0	632.4	69.3	1.76	112.9	632.4
Edtftpj-2.3.0	2.4.0	3	3	43.0	2.25	138.9	1116.3	69.9	2.30	139.2	632.4
Edtftpj-2.3.0	2.4.0	5	5	31.1	2.49	162.8	1600.2	69.9	2.42	163.9	632.4
Edtftpj-2.3.0	2.4.0	7	7	24.4	2.76	169.7	2084.0	69.9	2.61	206.1	632.4
Ganymed-ss2-build209	build210	1	2	76.0	1.31	105.6	512.2	76.0	1.66	93.2	512.2
Ganymed-ss2-build209	build210	3	3	51.4	1.83	120.3	846.2	76.0	1.91	119.2	512.2
Ganymed-ss2-build209	build210	5	5	38.8	2.40	156.5	1180.2	76.0	2.15	162.6	512.2
Ganymed-ss2-build209	build210	7	7	31.1	2.46	171.1	1514.1	76.0	2.27	168.5	512.2
Jsmpp-2.0	2.1	1	2	89.4	1.42	110.1	542.8	89.4	1.78	106.6	542.8
Jsmpp-2.0	2.1	3	3	73.7	1.84	128.2	711.7	89.4	1.93	120.5	542.8
Jsmpp-2.0	2.1	5	5	62.7	2.09	129.8	880.6	89.4	2.07	143.7	542.8
Jsmpp-2.0	2.1	7	7	54.6	2.19	164.4	1049.4	89.4	2.19	156.6	542.8
Kryonet-2.08	2.20	1	2	58.1	1.09	100.7	453.3	58.1	1.40	118.1	453.3
Kryonet-2.08	2.20	3	3	32.1	1.88	123.8	851.9	62.6	1.88	113.2	453.3
Kryonet-2.08	2.20	5	5	22.1	2.40	144.3	1250.4	62.6	2.20	158.8	453.3
Kryonet-2.08	2.20	7	7	16.9	2.61	160.8	1649.0	62.6	2.34	163.7	453.3
Mime4j-core-0.7.1	0.7.2	1	2	98.4	0.66	74.4	159.9	98.4	0.79	85.3	159.9
Mime4j-core-0.7.1	0.7.2	3	3	95.3	0.82	88.5	172.3	98.4	0.92	91.2	159.9
Mime4j-core-0.7.1	0.7.2	5	5	92.4	1.08	90.7	184.8	98.4	1.19	95.1	159.9
Mime4j-core-0.7.1	0.7.2	7	7	89.7	1.32	106.1	197.3	98.4	1.33	101.9	159.9
Xnio-2.0.0CR2	2.1.0CR1	1	2	51.4	0.98	101.7	501.1	51.4	1.24	98.9	501.1
Xnio-2.0.0CR2	2.1.0CR1	3	3	26.6	1.67	121.1	992.1	54.9	1.53	121.4	501.1
Xnio-2.0.0CR2	2.1.0CR1	5	5	17.9	2.04	132.1	1483.2	54.9	1.77	130.7	501.1
Xnio-2.0.0CR2	2.1.0CR1	7	7	13.4	2.62	171.5	1974.2	54.9	1.91	158.3	501.1
Netx-0.4	0.5	1	2	57.5	1.04	91.5	457.1	57.5	1.32	97.1	457.1
Netx-0.4	0.5	3	3	36.0	1.75	116.3	823.2	65.4	1.74	120.0	457.1
Netx-0.4	0.5	5	5	25.2	1.84	126.6	1189.2	65.4	1.75	125.0	457.1
Netx-0.4	0.5	7	7	19.4	2.07	156.2	1555.3	65.4	1.99	158.1	457.1

Table 3 Runtime performance comparison.

App.	Bytec. Size [KB]	#cl. v.1	#cl. v.2	Simple					Optimized					Without Centralization	
				Tran. Time [s]	Trans. Mem. [MB]	Exce. Time [s]	Exce. Mem. [MB]	Stora. [KB]	Tran. time [s]	Trans. Mem. [MB]	Exce. Time [s]	Exce. Mem. [MB]	Stora. [KB]	Exce. Time [s]	Exce. Mem. [MB]
Echo		1	1	0.37	33.59	0.14	22.79	5.35	0.40	36.93	0.14	22.77	5.35	0.27	64.08
Server	2.17	2	2	0.37	34.15	0.14	23.33	6.82	0.41	37.37	0.14	23.02	5.35	0.41	107.82
cl.v1	1.49	4	4	0.38	34.34	0.14	24.15	9.79	0.42	37.93	0.14	24.02	5.35	0.68	195.14
cl.v2	1.49	8	8	0.41	36.27	0.14	25.33	15.70	0.43	38.96	0.14	25.27	5.35	1.22	369.93
Daytime		1	1	0.37	33.24	0.15	24.28	3.90	0.40	36.45	0.19	24.36	3.90	0.36	66.03
Server	1.63	2	2	0.38	33.94	0.16	25.13	5.05	0.41	37.32	0.19	24.88	3.90	0.53	109.60
#cl.v.1	1.16	4	4	0.38	34.27	0.16	26.19	7.34	0.41	37.41	0.19	26.19	3.90	0.87	196.93
#cl.v.2	1.16	8	8	0.40	35.66	0.16	28.62	11.92	0.43	38.56	0.19	28.30	3.90	1.53	371.78
Chat		1	1	0.37	33.91	0.14	23.18	8.39	0.41	37.66	0.14	23.13	8.39	0.63	64.23
Server	3.99	2	2	0.38	34.23	0.15	24.18	10.56	0.42	37.96	0.14	23.91	8.39	0.66	108.30
#cl.v.1	1.98	4	4	0.40	36.09	0.15	25.62	14.93	0.44	38.98	0.14	25.38	8.39	0.72	196.85
#cl.v.2	1.98	8	8	0.44	44.62	0.17	30.12	23.64	0.46	46.50	0.17	28.49	8.39	0.96	373.75
Alphabet		1	1	0.38	35.82	0.42	23.41	8.51	0.42	39.14	0.45	23.53	8.51	0.57	64.60
Server	3.20	2	2	0.40	36.14	0.62	23.83	9.99	0.43	39.19	0.65	23.77	8.51	0.94	108.81
#cl.v.1	3.46	4	4	0.41	38.23	1.02	24.67	12.97	0.45	41.23	1.06	24.54	8.51	1.69	196.95
#cl.v.2	3.46	8	8	0.45	52.45	1.83	26.37	18.91	0.49	57.29	1.87	26.04	8.51	3.14	373.60

Server Client benchmark as an example. The size of its server is 2.17KB, and the size of each client version is 1.49KB. For each benchmark, we have four settings with different number of instances for each version of a client. The total benchmark size can therefore be calculated by taking size summation for all its projects. The transformation time and memory of cost for each algorithm are shown in columns *Trans. Time* and *Trans. Mem.*. Compared with benchmarks in Section 5.2, the benchmarks used in this section are relatively small and the time cost to output the centralized results is not an important factor. We find that the optimized centralization takes more time and memory to perform transformation.

The execution time and memory cost of the centralized application are shown in columns *Exec. Time* and *Exec. Mem.*. The execution time is the averaged real time for program execution and the memory consumption is the averaged peak memory consumption of the whole VM. They are measured by using GNU *time* tool. The storage refers to the memory cost (in KB) to store all class files of the centralized application. The execution time of the centralized applications does not show the significant difference for both transformation algorithms. The execution memory cost of the simple algorithm is slightly larger than the optimized solution because it produces more classes, which need to be loaded in to VM during runtime.

The execution time and memory of the original program are larger than for their centralized counterpart. Without centralization, each application runs on its own VM which produces additional overhead. Each VM also loads its own version of classes, many of which are duplicated among different applications. Therefore, the runtime and memory consumption grows linearly with the number of applications.

After centralization, the runtime behavior of the centralized distributed application can also be further analyzed by profiling tools like JRAT [8], JIP [19], which shows more detailed runtime behavior like the execution time of each centralized process, the time consumption of each method and so on. Centralization enables these tools to have a global view of the distributed system. As the centralized approach simulates the behavior of distributed application with less execution time and memory consumption, it could also be helpful for stress testing to find runtime bottlenecks earlier. We will leave this potential application of centralization as future work.

5.3 Centralization with JPF

In Section 1, we have discussed that centralization enables existing tools to analyze multi-process applications. This section presents our third experiment that applies centralization with Java PathFinder v6 to verify distributed applications

To show that our centralization tool performs a correct transformation, we repeat previous experiments using centralization with Java PathFinder (JPF) [1]. These experiments were ran on the Echo client/server, Daytime client/server, Chat Server and Alphabet client/server [2] as the test beds. These benchmarks are the same as the ones that we used in Section 5.2. We first apply the optimized centralization approach on these benchmarks. Each benchmark consists of one server and two clients with different versions. The verification results of JPF on the centralized application is summarized in **Table 4**. Column *Bytec. Size* shows the total size (including the server and two clients) of each benchmark, which can be calculated by summing up the size of each its application presented in Table 3. For a small application like the Echo client/server system, JPF finishes its verification in 7 seconds. Similarly, it takes about 1 minute for the Daytime case. The other applications are much more complex. The chat server features a high degree of internal concurrency because each request is sent back to all currently connected clients. The state space therefore contains all possibilities for concurrent client connections, with all possible permutations for establishing a connection, and for each message to be interleaved with other messages. Because of this, it takes more than one hour to finish searching the state space. The state space of Alphabet is even larger, because each client is implemented using producer and consumer threads. For the case with two active clients, this involves three threads for each application: on the server side, the main thread and two worker threads are used, and each client uses a main thread, a producer, and a consumer thread. Because the state space is exponential in the number of threads, this case is too large for JPF to handle: After 14 hours, it reports that it has run out of memory (given 1 GB of heap space). However, while full verification is out of reach for such applications, finding defects is still possible.

We cover that use case by seeding some faults into these benchmarks. Chat. v.1 is the buggy version that has a race condition on a shared array field that stores active connections. In the failure scenario, one client disconnects, causing the server worker thread handling that client to remove that entry. Because the "remove" operation is not synchronized, another worker thread (serving a different client) checks the contents of that field (which is non-null at first) before using it. Between the check and use, the unsynchronized remove operation sets the field to null, causing the NullPointerException in the other worker thread later. In Daytime LeapSecond, the server produces a time with leap second with low probability, and one client checks the format of the time it receives. The client crashes if the time format is incorrect. These two bugs could be found by JPF quickly. However, when a large number of threads is involved, it may take more time to find a defect in a large state space. The benchmarks of Alphabet. v.1 consists more than then threads (including a wrapper thread), and it takes more than 5 hours to find the seeded bug. Previous work [1] does not support centralizing distributed applications with multiple versions. The net-iocache approach [2] analyzes each peer separately, which cannot find these bugs, either. By using our centralization approach, we can successfully find these described bugs.

Table 4 Application of centralization to JPF.

App.	Bytec. Size [KB]	JPF.		Opt. Centra.	
		Time [h:mm:ss]	Mem. [MB]	Time [s]	Mem. [MB]
Echo Server Client	5.15	00:00:07	328	0.40	36.92
Chat Server Client	7.94	01:10:19	471	0.41	37.66
Chat Server Client v.1	7.94	00:00:01	82	0.42	37.95
Daytime Server Client	3.95	00:00:58	343	0.40	36.76
Daytime LeapSecond	6.13	00:00:14	366	0.41	37.00
Alphabet Server Client	10.11	N.A.	N.A.	0.43	39.20
Alphabet Server Client v.1	10.11	05:29:58	1023	0.42	39.28

6. Related Work

Stoller [17] initially proposes to use centralization for verifying distributed Java applications in Java PathFinder (JPF). Artho et al. [1] improve the accuracy of centralization and implement an automatic tool for such verification by JPF. However, the implementation uses the outdated SERP bytecode library [15], which makes it unable to work on current Java applications.

Compared with previous work, we intend to build an automatic centralization tool for 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 semantics does not depend on specific tools and transformations of static fields are also refined. Although the large state space of distributed applications limits software model checkers to small cases, our centralization approach enables existing dynamic analysis tools to analyze practical distributed applications.

Other work on verifying distributed applications in JPF includes net-iocache [2] and modeling the Java class loader [16]. Both solutions are specific to JPF. Compared with the centralization approach, net-iocache analyzes a single peer of a distributed application, which 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 [16]. It uses class loaders to separate process name spaces by a roundtrip collaboration between JPF and host VM. However, the cost of this roundtrip switch between JPF and host VM is expensive, which makes such approaches difficult to scale up for larger applications. Additional works on startup, shutdown behavior preservation, and modeling network library are also necessary to verify distributed applications. Currently, JPF v7 is under development. We will compare the class loader approach with our centralization approach after JPF v7 is released.

7. Conclusion and Future Work

In this paper, we have advanced centralization as a general analysis framework for distributed Java applications. We have formalized and solved the class version conflict problem to enable centralization for applications containing multiple versions of a given class. Based on the formalization, we have proposed two approaches: a simple solution and an optimized solution. We have also refined the centralization issues such as the startup order control of peers, and the transformation of static fields. We have implemented an automatic centralization tool and empirically evaluated proposed algorithms. The experiments show that optimized algorithm is more effective to share common code. It also shows that the centralized application has a more efficient runtime performance in terms of execution time and memory than its counterpart non-centralized one. The experiments using Java PathFinder demonstrate that centralization enables such an existing tool to analyze distributed applications, showing 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 such as deadlock and data race detectors, and visualization tools for analyzing distributed applications. For the code instrumentation, we plan to finish the remaining implementation of the shutdown semantics and extend our tool to support reflection by using renaming map.

Acknowledgments This work was partially supported by Global COE「Secure-Life Electronics」Program from the University of Tokyo, Japan.

References

[1] Artho, C. and Garoche, P.-L.: Accurate Centralization for Applying Model Checking on Networked Applications, *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, Washington, DC, USA, pp.177–188 (2006).

[2] Artho, C., Leungwattanakit, W., Hagiya, M., Tanabe, Y. and Yamamoto, M.: Cache-Based Model Checking of Networked Applications: From Linear to Branching Time, *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, Washington, DC, USA, pp.447–458 (2009).

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

[4] Dumitraş, T. and Narasimhan, P.: Why Do Upgrades Fail and What Can We Do About It?: Toward Dependable, Online Upgrades in Enterprise System, *Proc. 10th ACM/IFIP/USENIX International Conference on Middleware*, New York, NY, USA, pp.18:1–18:20 (2009).

[5] Dumitras, T., Narasimhan, P. and Tilevich, E.: To Upgrade or Not to Upgrade: Impact of Online Upgrades Across Multiple Administrative Domains, *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications* (*OOPSLA*), New York, NY, USA, pp.865–876 (2010).

[6] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. and Lea, D.: *Java Concurrency in Practice*, Addison-Wesley Professional (2006).

[7] James, G., Bill, J., Guy, L.J. and Gilad, B.: *Java Language Specification*, 3 edition, Addison-Wesley Professional (2005).

[8] Java Runtime Analysis Toolkit: available from ⟨http://jrat.sourceforge.net/⟩ (accessed 2013-02-13).

[9] JavaTM Platform, Standard Edition, V6 API Sepcification, available from ⟨http://docs.oracle.com/javase/6/docs/api/⟩ (accessed 2013-03-26).

[10] JCarder, available from ⟨http://www.jcarder.org/⟩ (accessed 2013-01-23).

[11] Jtracert, available from ⟨https://code.google.com/p/jtracert/⟩ (accessed 2013-03-13).

[12] Kuleshov, E.: Using ASM Framework to Implement Common Bytecode Transformation Patterns, *6th International Conference on Aspect-Oriented Software Development*, Vancouver, British Columbia (2007).

[13] Lehman, M. and Ramil, J.: Software Evolution in the Age of Component-based Software Engineering, *Software, IEE Proceedings*, Vol.147, No.6, pp.249–255 (2000).

[14] Letko, Z., Vojnar, T. and Křena, B.: AtomRace: Data Race and Atomicity Violation Detector and Healer, *Proc. 6th Workshop on Parallel and Distributed Systems: Testing, Snalysis, and Debugging*, New York, NY, USA, pp.7:1–7:10 (2008).

[15] Serp, available from ⟨http://serp.sourceforge.net/⟩ (accessed 2012-05-12).

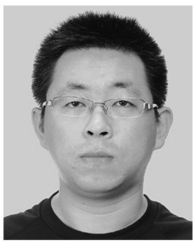
[16] Shafiei, N. and Mehlitz, P.: Modeling Class Loaders in Java PathFinder Version 7, *SIGSOFT Software Engineering Notes*, Vol.37, No.6, pp.1–5 (2012).

[17] Stoller, S.D. and Liu, Y.A.: Transformations for Model Checking Distributed Java Programs, *SPIN 2001*, NY, USA, pp.192–199, Springer-Verlag New York, Inc. (2001).

[18] Taniguchi, K., Ishio, T., Kamiya, T., Kusumoto, S. and Inoue, K.: Extracting Sequence Diagram from Execution Trace of Java Program, *Proc. 8th International Workshop on Principles of Software Evolution*, Washington, DC, USA, pp.148–154 (2005).

[19] The Java Interactive Profiler, available from ⟨http://jiprof.sourceforge.net/⟩ (accessed 2013-02-15).

[20] Visser, W., Havelund, K., Brat, G., Park, S. and Lerda, F.: Model Checking Programs, *Automated Software Engineering*, Vol.10, No.2, pp.203–232 (2003).



Lei Ma is a Ph.D. candidate at the Department of Electrical Engineering and Information Systems, The University of Tokyo. He received his B.S. from the Department of Computer Science, Shanghai Jiao Tong University in 2009, and M.S. from Department of Electrical Engineering and Information System, The University of Tokyo. His interests cover various topics related to programming languages, compiler optimization, type system, and software management and maintenance. His current research mainly focuses on software verification by formal techniques.

Cyrille Artho is a Research Scientist in AIST, Japan. His 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 in 2005. 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.

Hiroyuki Sato is an Associate Professor in The University of Tokyo. He received his B.Sc., M.Sc. and Ph.D. from The University Tokyo in 1985, 1987, 1990, respectively. He is majoring in Computer Science and Information Security.