Project Centralization Based on Graph Coloring

Lei Ma
Dept of Electrical Engineering and Information Systems
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

Version conflicts are common in a component-based system, where each component is developed and managed independently. Changes during the life-cycle of components require multiple versions to coexist. This creates a challenge in representing multiple versions for program analysis tools and execution platforms that are designed to handle only one version. In this paper, a project centralization approach is proposed to manage the version conflict problem. Our technique shares common code whenever possible while keeping the version space of each component separate. We formalize and transform the project centralization into a graph coloring problem. A corresponding algorithm is also presented. Experiments on real world software projects demonstrate the effectiveness of our technique.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Software management

Keywords

Version conflict, component-based system, distributed system, dynamic software update, program analysis

1. INTRODUCTION

Component-based software allows easier reuse and provides the flexibility of dynamic integration. A component with the same functionality can be developed by different groups with different implementations. Continuous change is necessary for a component to maintain its functionality and satisfy new requirements [14]. If multiple versions of a component interact, however, incompatibilities may cause a version conflict. This may happen when an application updates some of its components to a new version while other older versions of that component are still in use [17]. A version conflict also occurs when a single Java VM is used to run the distributed system consisting of multiple component-based applications, to reduce the runtime overhead and resource duplication [5]. Such an approach is also used for verifying and analyzing distributed applications [22, 1, 16].

Version conflicts get exacerbated in distributed systems, where installations are duplicated over many peers. Each component-based application is updated asynchronously in a "rolling update", creating multiple versions of the component in the system. Dumitraş et al. [7, 8] point out that most update failures are not caused by a software defect, but by version conflicts during the update procedure.

The analysis of multiple versions of software creates a challenge for many analysis tools and execution platforms of managed languages like Java and C#, which are designed to handle only one version. To analyze a distributed system in verification tools like Java PathFinder (JPF) [25], applications can be merged to run on a single virtual machine [22, 1, 16]. However, the Java Virtual Machine [15] does not allow loading multiple versions of a class that have the same name but different implementations in a single class loader, making such approaches unable to work for component-based applications with multiple versions [5].

Project centralization [16] is a general technique to manage component-based applications with multiple versions. It represents the code repositories of all components using a single code repository. Centralization shares common code whenever possible while keeping the version space of each component separate so that each transformed component exhibits the same run-time behavior as the original one.

In this paper, we formalize project centralization to resolve version conflicts for component-based applications. We first present our simple algorithm published earlier [16]. After discussing its limitations, we propose and formalize a graph-based representation for project centralization. Based on this, we transform the version separation problem into a graph coloring problem. Furthermore, we provide a corresponding algorithm to compute the optimal solution and the heuristic solution based on existing graph coloring algorithms. We evaluate the effectiveness of these algorithms in terms of storage and time by performing experiments on seven real-world Java projects. While our implementation supports Java bytecode [9], the concepts presented in this paper generalize to other managed programming languages and runtime platforms.

This paper is organized as follows. Section 2 formalizes project centralization and summarize our previous approach [16]. Section 3 elaborates our D-graph representation for projects and transforms project centralization into a graph coloring problem. Section 4 evaluates our proposed approaches. After discussing related work in Section 5, Section 6 concludes and proposes future work.

2. PROJECT CENTRALIZATION

The concurrent usage of different versions of a component is common in component-based systems. Project centralization resolves possible version conflicts by separating the version space of each component, while sharing common code among different systems. In this section, we formalize project centralization and discuss our previous project centralization algorithm [16] and its limitations.

2.1 Project Centralization Example

We will use a running example in this paper (see Fig. 1). Each project consists of a set of classes. A directed edge between two classes represents their dependency. For example, we draw a directed edge from class A to C in Project$_1$ because class C references A. In Fig. 1(a), Project$_1$ and Project$_2$ can share most of their classes except for C, where different versions are used. Compared to Project$_2$, Project$_3$ holds a different version of class *Main* and a new class *Unique*.

Project centralization transforms multiple projects into a single one, in which each project preserves its version space while sharing common code whenever possible. Fig. 1(b) shows the centralization result for projects in Fig. 1(a). All projects share class A. Project$_1$ renames its classes to $P_1.C$, $P_1.B$, and $P_1.Main$ to separate the version space. Similarly, Project$_2$ and Project$_3$ share classes C and B, and Project$_2$ renames its class *Main* to $P_2.Main$. Classes *Main* and *Unique* in Project$_3$ are left unchanged. The centralized result preserves the behavior of each project.

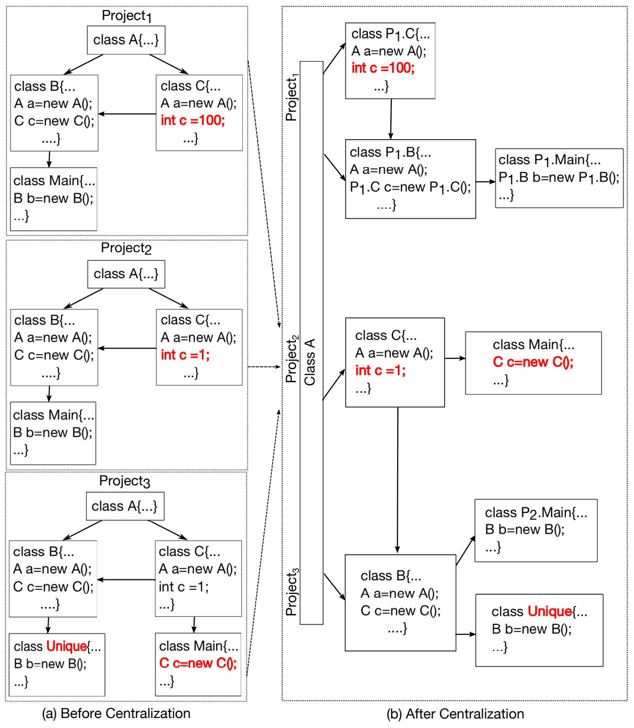
2.2 Formalization

A Java class is uniquely identified by its name (including package name) and implementation. For a class cl, we use $cl.name$ and $cl.code$ to denote its class name and implementation, respectively. Given two classes cl_1 and cl_2, cl_1 and cl_2 are equivalent, denoted by $cl_1 = cl_2$, if they form a **Type-1** clone pair [21], where $cl_1.name$ is identical to $cl_2.name$, and $cl_1.code$ and $cl_2.code$ are also identical except for variations in whitespace, layout and comments.

Definition 1. A *project* is a set of classes, in which each class has a distinct name. Given a project p, we write $\#p$ as the number of classes in p, and denote a class cl in p by $p.cl$. Two projects p and q are *identical*, if they hold the same set of classes.

A project represents an abstract view of the class repository of a component. Each component is represented by a project. Furthermore, the combination of all components can be represented by one *centralized* project by merging small projects. Two component-based applications may use code from either the same project or different projects. In both cases, code repositories of multiple components can be represented as a centralized project, sharing common code.

Definition 2. Let p be a project. We define $\text{NAME}(p) = \{cl.name | cl \in p\}$ as the set that contains all class names in p. For a class name $cln \in \text{NAME}(p)$, we define GetClass(p, cln) $= p.cl$, where $p.cl.name = cln$, as a function to get the class named cln in p. Let P be a set of projects. We define



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
Project3
Class A
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

Figure 1: Project Centralization Example

$\text{NAMES}(P) = \cup_{p \in P}\text{NAME}(p)$ as the set containing all the class names in P, and $P{\uparrow}cln = \{p \in P | cln \in \text{NAME}(p)\}$ as the set of all 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 p substitutes its class name cln_1 for cln_2. Substitution includes class names and references to them. A renaming substitution $p[cln_1/cln_2]$ is a *normal substitution* if $cln_1 \notin \text{NAME}(p)$ and $cln_2 \in \text{NAME}(p)$.

Definition 4. Let p_1 and p_2 be two projects; p_1 is equivalent to p_2, denoted by $p_1 = p_2$, if they can be renamed to identical projects by normal substitutions. It is not difficult to prove that this is an *equivalence* relation.

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

Project centralization requires preservation of the class version space for each project. Each component-based application that runs as the original project can also run as the centralized project with the same runtime behavior. The projects to be centralized can either be different versions of a component or different components.

Different from project centralization, *process centralization* [22, 5, 1] simulates the runtime behavior of multiple component-based applications by a single application with equivalent runtime behavior. Process centralization does not cover multiple versions of the same component. It assumes that each class has only one version and no version conflict exists. On the other hand, project centralization shares common code of component-based applications to save storage while keeping the version space of each application separate to avoid version conflicts. Project centralization enables process centralization for component-based applications with multiple versions of the same component.

Definition 6. Let cl_1 and cl_2 be two classes in a project p. Class cl_1 depends on cl_2, denoted by $cl_2 \rightarrow cl_1$ if $cl_1.code$ references $cl_2.name$.

The class dependency represents the class reference relation in a project. If two classes cl_1,cl_2 have a dependency relation $cl_1 \rightarrow cl_2$ and cl_1 is renamed, all references of cl_1 in cl_2 must also be renamed to preserve the implementation.

Let P be a set of projects to be centralized. We classify the classes of a project $p \in P$ into the following categories:

1. *Unique Class.* $\text{UNIQUE}(p, P) = \{cl \in p | \forall q \in P \backslash p.\ cl.name \notin \text{NAME}(q)\}$. A unique class of project $p \in P$ has a unique name across all projects in P.
2. *Conflict Class.* $\text{CONFLICT}(p, P) = \{cl \in p | \exists q \in P.\ cl.name \in (\text{NAME}(p) \cap \text{NAME}(q)) \wedge p.cl \neq \text{GetClass}(q, cl.name)\}$. The name of a conflict class appears in multiple projects, but with different implementations.
3. *Shared Class.* $\text{SHARED}(p, P) = \{cl \in p | \exists q \in P \backslash p.\ cl.name \in (\text{NAME}(p) \cap \text{NAME}(q)) \wedge p.cl = \text{GetClass}(q, cl.name)\}$. A shared class of p shares both its name and implementation with other projects in P.

In our example in Fig. 1(a), classes A and B are shared classes in all projects. The cases for classes C and *Main* are more complex: class C is a conflict class in Project_1, but it is both shared and a conflict class in Project_2 and Project_3. Similarly, class *Main* is a conflict class in Project_3, and it is both shared and a conflict class in Project_1 and Project_2.

Definition 7. Let P be a set of projects to be centralized. Centralized project p_{centr} is minimal (optimal) if $p_{centr} \in \text{CENTR}(P)$ and $\forall p'_{centr} \in \text{CENTR}(P).\ \#p_{centr} \leq \#p'_{centr}$.

Consider a general scenario of centralizing a set of projects $P = \{p_1, p_2, \ldots, p_n\}$. There may exist multiple solutions that satisfy *Definition* 5. Among these solutions, the optimal solution outputs the minimal number of classes. If $\forall p_i \in P.\, \text{CONFLICT}(p_i, P) = \emptyset$, the optimal result is the union of all projects in P, $\bigcup_{i=1}^{n} p_i$. If conflicts are present, $\exists p \in P.\, \text{CONFLICT}(p, P) \neq \emptyset$, the goal is to separate all conflict classes in P while maximizing the sharing of classes.

2.3 Simple Algorithm and its Limitation

Separation of the class version for each project entails renaming the conflict classes and all their references. However, such renaming may cause shared classes not to be shareable anymore, as their internal references to other classes are renamed differently across projects. Consider our example in Fig. 1(a): Class B in Project_1 and Project_2 is identical and could be shared as such. However, project centralization renames class C in these projects to a different name to resolve a version conflict. After that step, B cannot be shared anymore as it references C. Therefore, effects of renaming conflict classes propagate through each project.

Given n projects containing m class names in total, our previous algorithm [16] renames all the conflict classes of the first $n-1$ projects and propagates the renaming effect by traversing class dependency relations of each project. Its complexity is $\mathcal{O}(m^2 * n)$ and it does not always output an optimal solution. The previous algorithm does not distinguish between a conflict class and a class that is both shared and in conflict. It simply renames each class that is a conflict class. Consider a set of project $P = \{p_1, p_2, p_4, p_3\}$ for centralization, where each project $p_i \in P$ has one class A; p_1, p_2 share one version of A and p_3, p_4 share another version. The simple renaming algorithm renames all A in p_1, p_2, p_3, but not in p_4, resulting in three classes. However, the optimal solution produces only two classes: one is shared by p_1 and p_2 and the other one is shared by p_3 and p_4.

3. GRAPH COLORING BASED APPROACH

Project centralization is an optimization problem. The goal is to obtain a centralized project with the minimal number of classes under given version constraints. We formalize the *D-graph* representation for projects and transform project centralization into a graph coloring problem. Then, we present a corresponding algorithm based on existing graph coloring solutions. In the rest this section, we arbitrarily fix a set of projects P for centralization.

3.1 Constraint Graph, Constraint Structure

A *constraint graph* represents the version relation of all classes with the same name in P. We show that all constraint graphs of a class name in P form a complete lattice, and extend the constraint graph to a *constraint structure*, which represents a node of a *D-graph*, as defined below.

Definition 8. Let *cln* be a class name in P. A constraint graph of *cln* in P consists of a pair of node set $P \uparrow cln$ and edge set CE, denoted by $\langle P \uparrow cln\ , CE \rangle$, such that if there exist two projects $p, p' \in P \uparrow cln$ and $(p, p') \in CE$, p and p' cannot share the class named *cln*.

Each node in a constraint graph of name *cln* is a project containing a class named *cln*. Two project nodes that are connected by an edge, cannot share the same version of the class named *cln*. Edges in a constraint graph are undirected; given any two project nodes m and n, (m, n) and (n, m) represent the same edge. Let $G = \langle P \uparrow cln\ , CE \rangle$ be a constraint graph of *cln* in P, and P' be a project set. We write the subgraph of G to P' as $\langle P'', CE' \rangle$ (denoted by $G \Uparrow P'$), where $P'' = P' \cap P$, and CE' is a restriction of CE to P''.

We define the partial order relations over constraint graphs and denote the constraint graph domain of P by $\mathbb{CG}$.

Definition 9. We define $\sqsubseteq_{cg} \in \mathbb{CG} \times \mathbb{CG}$ as a binary relation such that for any two constraint graphs $G_1, G_2 \in \mathbb{CG}$ with $G_1 = \langle P_1, CE_1 \rangle$ and $G_2 = \langle P_2, CE_2 \rangle$, $G_1 \sqsubseteq_{cg} G_2$ if $P_1 \subseteq P_2$ and $CE_1 \subseteq CE_2$. We denote the least upper bound of G_1 and G_2 by $G_1 \sqcup_{cg} G_2 = \langle P_1 \cup P_2, CE_1 \cup CE_2 \rangle$.

It is not difficult to prove that $\sqsubseteq_{cg} \in \mathbb{CG} \times \mathbb{CG}$ is a partial order and $(\mathbb{CG}, \sqsubseteq_{cg})$ is a partially ordered set.

Let *cln* be a class name in P; we write $\mathbb{CG}_{cln}$ as the subdomain of $\mathbb{CG}$, where $\mathbb{CG}_{cln} \subseteq \mathbb{CG}$ and $\forall G \in \mathbb{CG}_{cln}.\, G = \langle P', CE' \rangle \Rightarrow P' = P \uparrow cln$. It is not difficult to prove that the partially ordered set $(\mathbb{CG}_{cln}, \sqsubseteq_{cg}, \sqcup_{cg}, \sqcap_{cg}, \perp_{cg}^{cln}, \top_{cg}^{cln})$ is a complete lattice [2] with, $\forall X \subseteq \mathbb{G}$, $X = \{x_1, x_2, \ldots, x_n\}$:

- a least upper bound $\sqcup_{cg} X = x_1 \sqcup_{cg} x_2 \sqcup_{cg} \ldots \sqcup_{cg} x_n$,
- a greatest lower bound $\sqcap_{cg} X = \sqcup_{cg} \{y | \forall x \in X.\, y \sqsubseteq_{cg} x\}$,

- a least element $\bot_{cg}^{cln} = \langle P\uparrow cln\ , \emptyset\rangle$,
- a greatest element $\top_{cg}^{cln} = \langle P\uparrow cln\ , CE_{greatest}\rangle$, where $CE_{greatest} = \{(m,n)|n,m \in\ P\uparrow cln \wedge m \neq n\}$.

We extend the constraint graph to a *constraint structure*.

Definition 10. A *constraint structure* CS in a project set P consists of a name in NAMES(P) (denoted by $CS.name$) and a constraint graph of the name $CS.name$ (denoted by $CS.CG$). We write $\langle CS.name, CS.CG\rangle$ for the structure.

We define the partial order relation and least upper bound for constraint structures.

Definition 11. Let CS_1 and CS_2 be two constraint structures. We define the partial order relation (constructed from the partial order of the constraint graph) $CS_1 \sqsubseteq_{cs} CS_2$ if $CS_1.name = CS_2.name$ and $CS_1.CG \sqsubseteq_{cg} CS_2.CG$. The least upper bound of CS_1 and CS_2 is defined as $CS_1 \sqcup_{cs} CS_2 = \langle CS_1.name,\ CS_1.CG \sqcup_{cg} CS_2.CG\rangle$ if $CS_1.name = CS_2.name$.

For a class name cln in P, it can be shown that all constraint structures that share cln also form a complete lattice.

3.2 D-graph Representation of a Project Set

We formalize the *D-graph* representation for projects, and propose constraint equations to calculate the minimal D-graph that satisfies version constraints. We then transform project centralization into a graph coloring problem.

Definition 12. A *D-graph* of a set of projects P consists a node set N of constraint structures and an edge set E (denoted by $\langle N, E\rangle$) with each edge $e = (l, m) \in E$ associated with a set of projects $e.set = \{p \in (P\uparrow l.name \cap P\uparrow m.name\,)\,|\text{GetClass}(p, l.name) \rightarrow \text{GetClass}(p, m.name)\}$ such that:

1. Name set $\{n.name|n \in N\}$ is the same as NAMES(P).
2. $\forall i, j \in N. e = (i, j) \wedge e.set \neq \emptyset \Rightarrow e \in E$.

Let $G = \langle N, E\rangle$ be a D-graph of P. Each node $n \in N$ is a constraint structure that represents all versions of the classes with name $n.name$. Its constraint graph $n.CG = \langle P\uparrow n.name\ , CE\rangle$ keeps the version relation of these classes. We denote the *predecessor* of a node $n \in N$ in G by $\text{Pred}(n) = \{m|(m,n) \in E\}$. For two nodes $m, n \in N$, the existence of an edge (m, n) from m to n entails that the classes named $m.name$ and $n.name$ have a dependency relation in a project p, and p occurs in both $P\uparrow m.name$ and $P\uparrow n.name$. Edges in E are directed: (n, m) and (m, n) are different edges.

Fig. 2(a) gives the corresponding initial D-graph of the project set in Fig. 1(a). The larger node is the constraint structure node, inside which its name and constraint graph are shown. For example, the node named A with its constraint graph indicates that its name exists in projects P_1, P_2 and P_3. No edge exists between these projects, meaning all these projects initially have the same version of class named A. The label of an edge in a D-graph shows the projects in which the two constraint structure nodes connected by that edge have a dependency relation. 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.

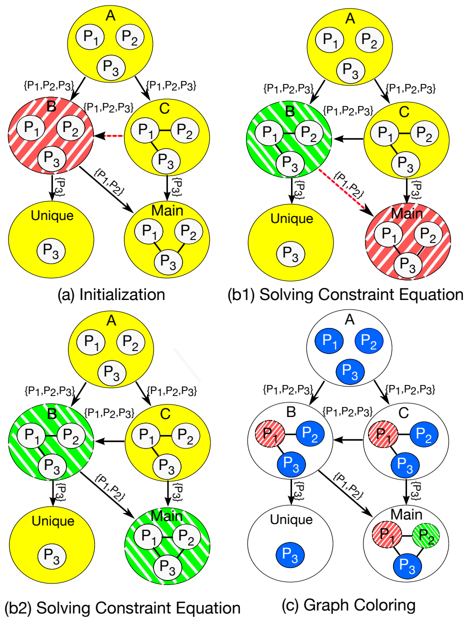


Figure 2: D-graph Representation Example

Definition 13. Let $G = \langle N, E\rangle$ be a D-graph of P. We define its *underlying graph* as the graph of G by ignoring the constraint graph of each constraint structure node in N, denoted by $|G| = \langle |N|, E\rangle$, where $|N|$ represents the nodes of G that ignore all their constraint graphs.

The underlying graph $|G|$ for a project set P is unique. There are multiple D-graphs that share $|G|$, differing in their constraint structures. We use $\mathbb{G}_{|G|}$ to represent the domain all D-graphs of P such that they share $|G|$ as the underlying graph. We simply write $\mathbb{G}$ if $|G|$ is clear from context. We continue to define a partial order over $\mathbb{G}$ and show that all its D-graphs also form a complete lattice.

Definition 14. Let $\mathbb{G}$ be a D-graph domain of P, and $G = \langle N, E\rangle$ and $G' = \langle N', E\rangle$ be arbitrary two D-graphs in $\mathbb{G}$. G and G' have the binary relation $G \sqsubseteq G'$ if and $\forall n \in N. \forall n' \in N'. n.name = n'.name \Rightarrow n \sqsubseteq_{cs} n'$. The least upper bound of G and G' is $G \sqcup G' = \langle N'', E\rangle$, where $N'' = \{m \sqcup_{cs} n | n \in N \wedge m \in N' \wedge m.name = n.name\}$.

Assuming NAMES(P) $= \{cln_1, cln_2, \ldots, cln_k\}$, the constraint graph domain and underlying graph of P be $\mathbb{G}$ and $|G| = \langle |N|, E\rangle$, respectively. It is not difficult to prove that $(\mathbb{G}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a complete lattice [2] such that $\forall X \subseteq \mathbb{G}$ where $X = \{x_1, x_2, \ldots, x_n\}$ with:

- a least upper bound $\sqcup X = x_1 \sqcup x_2 \sqcup \ldots \sqcup x_n$,
- a greatest lower bound $\sqcap X = \sqcup\{y|\forall x \in X. y \sqsubseteq x\}$,
- a least element $\bot = \langle\{\langle cln_1, \bot_{cg}^{cln_1}\rangle, \langle cln_2, \bot_{cg}^{cln_2}\rangle, \ldots, \langle cln_k, \bot_{cg}^{cln_k}\rangle\}, E\rangle$,
- a greatest element $\top = \langle\{\langle cln_1, \top_{cg}^{cln_1}\rangle, \langle cln_2, \bot_{cg}^{cln_2}\rangle, \ldots, \langle cln_k, \top_{cg}^{cln_k}\rangle\}, E\rangle$.

Correct project centralization requires separating the version spaces of each project by renaming. Renaming a class also entails renaming all references to it accordingly. We use *conflict edges* to represent version constraint conditions. Such constraints 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 15. Let $G = \langle N, E\rangle$ be a D-graph of P. Let $e \in E$ be an edge and $e = (m, n)$, where $m.CG = \langle P \uparrow m.name\ , CE\rangle$ and $n.CG = \langle P \uparrow n.name\ , CE'\rangle$. The edge e is a *conflict edge* if $\exists p, q \in e.set.\ (p, q) \in CE \wedge (p, q) \notin CE'$.

An edge $e = (m, n)$ in a D-graph of P is a conflict edge, if there exist two projects $p, p' \in e.set$ such that they are connected by an edge in $m.CG$ but not in $n.CG$. For example, the dashed edge from node C to B in Fig. 2(a) is a conflict edge. P_1 and P_3 must hold a different version of class C, as they are connected by an edge in $C.CG$. This entails renaming their C to a different name; furthermore, as C is referenced in the code of B, P_1 and P_3 must also be connected in $B.CG$ to separate version space.

Definition 16. A D-graph $G = \langle N, E\rangle$ of P is *valid* if there does not exist an edge $e \in E$ such that e is a conflict edge.

Correct project centralization requires finding a valid D-graph given a set of projects, such that all version constraints are resolved. An optimal solution requires a D-graph that is both valid and minimal. This entails propagating the minimal version constraints so that each constraint structure node n in that D-graph satisfies the equations in (1), where IN(n) and OUT(n) are the incoming and outgoing constraint conditions (represented by the constraint graphs) of node n, and $\mathrm{IN}_0(n)$ and $\mathrm{OUT}_0(n)$ are the corresponding initial conditions, respectively. The functions in equational system (1) are monotonically increasing over complete lattices with finite height. Therefore, a minimal solution exists and can be computed by iterating the equation system [24].

$$\boxed{\begin{aligned}&\mathrm{IN}(n) = \Big(\bigsqcup_{cg\atop m\in \mathrm{Pred}(n)} \mathrm{OUT}(m)\Big) \uparrow (m, n).set\\ &\mathrm{OUT}(n) = \mathrm{IN}(n) \sqcup \mathrm{OUT}(n)\\ &\mathrm{IN}_0(n) = \emptyset\\ &\mathrm{OUT}_0(n) = n.CG\end{aligned}} \qquad (1)$$

Initially, the incoming constraint for each node $n \in N$ is empty and the outgoing constraint equals $n.CG$. The constraint graph $n.CG$ is initialized during the construction of the D-graph such that two project nodes are connected by an edge in $n.CG$ if they hold a different version of the class named $n.name$. The initial D-graph of the example in Fig. 1(a) is depicted in Fig. 2(a). The constraint equations are solved iteratively until no conflict edge exists. Fig. 2(b1) and Fig. 2(b2) show such steps to solve constraint equations for node B and $Main$, respectively. The minimal valid result is given in Fig. 2(c).

The remaining task is to ensure no two nodes connected by an edge in a constraint graph of the minimal valid D-graph share the same version. This is equivalent to coloring the graph such that two nodes connected by an edge are colored differently. After coloring, all the nodes in a constraint graph with the same color can share the same version of a class, and nodes colored differently cannot share the same class and should be renamed accordingly. In our example, all three projects share class A; class B outputs two versions, one of which is for P_1 and the other version is shared by P_2 and P_3 as shown in Fig. 2(c).

Algorithm 1 Graph Coloring Based Project Centralization

```
 1: procedure ProjectCentralization
Input: A set of projects $P = \{p_1, p_2, \ldots, p_n\}$
Output: The centralized project $p_{centr}$,
        where $\forall p \in P. \exists p' \subseteq p_{centr}. p = p'$
 2:     $DGraph \leftarrow \emptyset$
 3:     $nameSet \leftarrow$ collectName($P$)        ▷ Collect all class names
 4:     $DGraph.nodeSet \leftarrow \emptyset$
 5:     for all $name \in nameSet$ do     ▷ Build a node for each name
 6:         $DGraph.nodeSet \leftarrow DGraph.nodeSet$
                          $\cup$ {createNode($name, P$)}
 7:     end for
 8:     for all $src \in DGraph.nodeSet$ do              ▷ Add edges
 9:         for all $targ \in DGraph.nodeSet \backslash src$ do
10:             $tempSet \leftarrow \{p | p \in (P \uparrow src.name \cap P \uparrow targ.name)$
11:             $\wedge$GetClass($p, src.name$) $\rightarrow$ GetClass($p, targ.name$)}
12:             if $tempSet \neq \emptyset$ then
13:                 $(src, targ).set = tempSet$
14:                 $Dgraph.edgeSet \leftarrow Dgraph.edgeSet \cup \{(src, targ)\}$
15:             end if
16:         end for
17:     end for
18:     Initialize IN($n$) and OUT($n$) for each $n \in DGraph.nodeSet$
19:     $SCCs \leftarrow$ calculateSCC($DGraph$)
20:     $TopoSCCs \leftarrow$ calculateTopologicalOrder($SCCs$)
21:     $Dgraph \leftarrow$ EquationSolver($Dgraph, TopoSCCs$)     ▷ Alg. 2
      solves constraint equations until reaching the least fixed point
22:     for all $node \in DGraph.nodeSet$ do
23:         graphColoring($node.CG$)
          ▷ Color each output constraint graph by existing algorithm
24:     end for
25:     $p_{centr} \leftarrow$ NormalRenaming($DGraph$)
      ▷ Perform normal substitution according to the coloring results
26:     return $p_{centr}$
27: end procedure
```

3.3 Algorithm and Optimal Solution

Based on the D-graph representation for a set of projects, obtaining the project centralization solution for version separation entails the following steps:

1. Solve the constraint equation for each node to get the minimal valid D-graph.
2. Color the constraint graph in each constraint structure node n of the D-graph such that any two nodes connected by an edge in $n.CG$ are colored differently.
3. Perform normal renaming substitution for each constraint graph such that project nodes with the same color still share the same class after renaming while nodes with different colors do not.

We propose a project centralization algorithm based on graph coloring (see Alg. 1). Given a set of projects as input, the algorithm first initializes a D-graph (lines 2–17) and the IN and OUT constraints for each of its nodes. To improve convergence towards the fix point, we calculate all Strongly Connected Components (SCC) and sort them in a topological order. Next, function EquationSolver (see Alg. 2) is called to solve the constraint equations iteratively for the ordered nodes until reaching the least fixed point (lines 18–21). The last step colors the constraint graph of each node and performs normal renaming substitution (lines 22–25).

To analyze the complexity of our algorithm, we assume the D-graph is initialized and it is only necessary to calculate the renaming decision for further processing. The D-graph initialization (lines 2–17) and renaming substitution (line 25) are specific to the given projects and operating system, so we do not consider them in the complexity analysis.

Let P be the input projects with $\#P = n$, and its initial D-graph be $G = \langle N, E\rangle$ with $\#N = m$ and $\#E = l$.

Algorithm 2 Solve Constraint Equations

```
1: function EQUATIONSOLVER
        ▷ Solve constraints for a given graph in SCCs' topological order
Input: $graph$: a D-graph,
       $TopoSCCs$: the topological order of SCCs for $graph$
Output: $graph$: the minimal valid D-graph
2:   for $SCC \in TopoSCCs$ do
                                   ▷ Visit each $SCC$ in topological order
3:     repeat ▷ Repeat if constraints of a node in $SCC$ change
4:       for $n \in graph.nodeSet \wedge n \in SCC$ do
5:         $\mathrm{IN}(n) \leftarrow (\bigsqcup_{m \in \mathrm{Pred}(n)} \mathrm{OUT}(m)) \uparrow (m, n).set$
6:         $\mathrm{OUT}(n) \leftarrow \mathrm{IN}(n) \sqcup \mathrm{OUT}(n)$
7:         $n.CG \leftarrow \mathrm{OUT}(n)$
8:       end for
9:     until $\forall n \in (graph.nodeSet \cap SCC).\, \mathrm{IN}(n)$ and
             $\mathrm{OUT}(n)$ do not change
10:  end for
        ▷ Function terminates when no constraint conditions change
11:  return graph
12: end function
```

Alg. 1 is guaranteed to terminate. It iteratively solves constraint equations for each node of G in order until reaching a fix point. As the constraint functions in Alg. 2 are monotonically increasing over a complete lattice with finite height, the least fixed point can be reached in no more than $n^2 \cdot m$ iterations. The complexity of computing the constraint conditions for a node is $\mathcal{O}(n^2 \cdot m)$. Computing the SCCs and their topological cost $\mathcal{O}(m \cdot l)$. Assuming the complexity of adopted graph coloring algorithm for a k-vertex and t-edge graph is $\alpha(k, t)$, the total complexity of solving a given D-graph and making the renaming decision is $\alpha(n, l) \cdot m + \mathcal{O}(n^4 \cdot m^3)$. The complexity of the iterative framework analysis depends on the graph structure of a given D-graph. Proving a tighter upper bound for complexity is beyond the scope of this paper and is future work.

To obtain the optimal project centralization result, it is necessary to apply an exact graph coloring algorithm on the achieved minimal D-graph so that the number of classes in the output is minimal. We adopt an existing optimal algorithm [3], which solves the problem of a k-vertex graph in PSPACE and in time $\mathcal{O}(5.283^k)$. However, the exact graph coloring is an NP-complete problem and its complexity is exponential. Therefore, we also provide the option to use the Greedy Independent Sets (GIS) coloring approach [13] with complexity $\mathcal{O}(k \cdot v)$, where k and v represent the number of vertexes and edges, respectively.

Finally, when using the resulting classes in Java, reflection [9] is widely used to dynamically load a class. For example, Java library methods like ClassLoader.loadClass load a class with a given name at runtime. To support dynamic class loading by reflection after renaming, we keep a renaming map for each project to track the renaming decisions made after graph coloring. We instrument application code before several key functions that load a class by reflection. If the class name to be loaded is included in the renaming map, then we use its corresponding new class name.

4. EXPERIMENTS

We have implemented the proposed algorithms in Java and applied them on seven real-world Java projects as benchmarks. Our implementation transforms the Java bytecode of the target applications. As our tool does not require the source code of the application, it also works for languages other than Java that compile to Java bytecode. Table 1 summarizes the benchmarks, where the project size and number of classes are listed. All experiments were run on an Intel Core i7 Mac 2.4 GHz with 8 GB of RAM, running Mac OS X 10.8.3 and Oracle's Java VM, version 1.7.0_21.

To quantify and compare the effectiveness of each algorithm in sharing common code, we define *Shared* as the ratio of shared classes to output classes: $Shared = \frac{\#ClassShared}{\#OutputClass}$. A class named *cln* is counted as shared if at least two projects share that class in the renaming decision. Consider the graph coloring results in Fig. 2(c), classes A, B, C are shared because multiple projects are colored the same in their constraint graph, but class *Unique* and *Main* are not. *Shared* ranges from 0 to 1; the larger its value, the more classes are shared. The trivial renaming approach renames all classes of each projects and shares no classes, *Shared* is therefore 0.

We run the experiment to centralize projects of each benchmark with a different number of instances per version. Each experiment is repeated 60 times to collect the *Shared* value, run time, and storage saving ratio (project size before centralization/after centralization). After choosing a benchmark with a specific setting, the *Shared* value and storage ratio of a given project centralization algorithm are unique. As for the run time, we discard the data of the first 10 runs, which may be influenced by disk I/O to read the input, and take the average of the other results.

The overall experimental result is summarized in Table 2. We have five experimental settings for each benchmarks with two versions of a project, from centralizing one and two project instances, to seven instances of both versions. The performance of the three approaches for each setting is listed in columns *Shared*, Storage Ratio, and Time, respectively, which can be compared horizontally and vertically. For each experimental setting, the greedy approach outperforms the simple solution and performs as well as the optimal solution in sharing common code and saving storage, as shown by *Shared* value and the storage ratio. As the number of project instances for centralization increases in each benchmark, the *Shared* value of the simple algorithm decreases. It indicates that some classes are not sharable by the simple solution when centralizing more class instances, but they still can be shared by the greedy solution and the optimal solution. The storage saving ratio of each approach increases as there is a growth in the number of project instances. Compared with the simple solution, the greedy solution and optimal solution both have a larger value of storage saving ratio, meaning that they saves more storage than the simple solution. As for run time, the simple algorithm is the most efficient one, and the optimal algorithm does not scale. Consistent with the complexity analysis in Section 3, the run time of the optimal algorithm grows exponentially in the number of projects in Table 2. The optimal algorithm cannot solve larger settings in a reasonable time (1 hour). Compared with the other two approaches, the greedy centralization is effective in sharing common code and efficient enough in practice.

5. RELATED WORK

Hnetynka et al. [11] originally discussed the component version conflict problem in Java component-based systems. They adopt the renaming approach by augmenting the class name of each component with a version identifier during dynamic class loading. This solution is equivalent to trivial renaming. As our algorithm maximizes the sharing of common classes, our solution requires less memory to represent the transformed code. Another approach adopts a modi-

Table 1: Summarization of Bechmarks

Project name / version	Edtftp		Ganymed-ss2		Jsmpp		Kryonet		Mime4j-core		Xnio		Netx	
	2.3.0	2.4.0	build209	build210	2.0	2.1	2.08	2.20	0.7.1	0.7.2	2.0.0CR2	2.1.0CR1	0.4	0.5
Bytecode size[KB]	352	391	305	345	457	458	206	252	154	154	249	254	240	246
#Cl. (*.class)	106	113	115	133	201	202	79	104	61	61	72	74	91	88

Table 2: Experimental Results of Project Centralization

Project name / version		Inst.		Shared [%]			Strorage Ratio			Time [ms]		
				Simple	Greedy	Optimal	Simple	Greedy	Optimal	Simple	Greedy	Optimal
Edtftpj-2.3.0	2.4.0	1	2	69.3	69.3	69.3	1.79	1.79	1.79	1.07	7.65	10.65
Edtftpj-2.3.0	2.4.0	2	2	53.1	69.9	69.9	1.70	2.35	2.35	1.51	9.71	51.00
Edtftpj-2.3.0	2.4.0	3	3	43.0	69.9	69.9	2.00	3.52	3.52	2.80	11.72	139.06
Edtftpj-2.3.0	2.4.0	5	5	31.1	69.9	69.9	2.32	5.87	5.87	6.77	20.70	5349.42
Edtftpj-2.3.0	2.4.0	7	7	24.4	69.9	N.A.	2.50	8.22	N.A.	12.60	32.30	> 1 h
Ganymed-ss2-build209	build210	1	2	76.0	76.0	76.0	1.94	1.94	1.94	1.02	8.85	12.02
Ganymed-ss2-build209	build210	2	2	61.3	76.0	76.0	1.91	2.54	2.54	1.38	9.65	52.97
Ganymed-ss2-build209	build210	3	3	51.4	76.0	76.0	2.30	3.81	3.81	2.60	12.49	138.04
Ganymed-ss2-build209	build210	5	5	38.8	76.0	76.0	2.75	6.35	6.35	6.39	20.06	5073.64
Ganymed-ss2-build209	build210	7	7	31.1	76.0	N.A.	3.01	8.88	N.A.	12.16	32.13	> 1 h
Jsmpp-2.0	2.1	1	2	89.4	89.4	89.4	2.53	2.53	2.53	1.18	17.68	23.13
Jsmpp-2.0	2.1	2	2	80.8	89.4	89.4	2.92	3.37	3.37	1.88	20.79	83.28
Jsmpp-2.0	2.1	3	3	73.7	89.4	89.4	3.86	5.06	5.06	3.99	28.39	188.76
Jsmpp-2.0	2.1	5	5	62.7	89.4	89.4	5.20	8.43	8.43	10.79	44.76	7109.86
Jsmpp-2.0	2.1	7	7	54.6	89.4	N.A.	6.11	11.80	N.A.	21.67	67.16	> 1 h
Kryonet-2.08	2.20	1	2	58.1	58.1	58.1	1.56	1.56	1.56	0.95	7.45	17.48
Kryonet-2.08	2.20	2	2	41.5	62.6	62.6	1.40	2.02	2.02	1.48	9.83	48.52
Kryonet-2.08	2.20	3	3	32.1	62.6	62.6	1.61	3.02	3.02	2.42	13.40	144.54
Kryonet-2.08	2.20	5	5	22.1	62.6	62.6	1.83	5.04	5.04	4.55	26.39	5119.54
Kryonet-2.08	2.20	7	7	16.9	62.6	N.A.	1.94	7.06	N.A.	8.59	42.94	> 1 h
Mime4j-core-0.7.1	0.7.2	1	2	98.4	98.4	98.4	2.88	2.88	2.88	0.84	3.89	5.74
Mime4j-core-0.7.1	0.7.2	2	2	96.8	98.4	98.4	3.70	3.84	3.84	0.92	4.41	24.94
Mime4j-core-0.7.1	0.7.2	3	3	95.3	98.4	98.4	5.35	5.77	5.77	1.58	5.32	45.27
Mime4j-core-0.7.1	0.7.2	5	5	92.4	98.4	98.4	8.31	9.61	9.61	3.56	6.72	1797.52
Mime4j-core-0.7.1	0.7.2	7	7	89.7	98.4	N.A.	10.90	13.45	N.A.	6.65	10.6	> 1 h
Netx-0.4	0.5	1	2	57.5	57.5	57.5	1.60	1.60	1.60	0.98	8.63	15.46
Netx-0.4	0.5	2	2	45.7	65.4	65.4	1.52	2.13	2.13	1.34	8.68	45.49
Netx-0.4	0.5	3	3	36.0	65.4	65.4	1.77	3.19	3.19	2.18	10.20	130.22
Netx-0.4	0.5	5	5	25.2	65.4	65.4	2.05	5.32	5.32	4.49	18.02	4790.91
Netx-0.4	0.5	7	7	19.4	65.4	N.A.	2.19	7.45	N.A.	8.28	29.10	> 1 h
Xnio-2.0.0CR2	2.1.0CR1	1	2	51.4	51.4	51.4	1.51	1.51	1.51	1.43	4.28	12.33
Xnio-2.0.0CR2	2.1.0CR1	2	2	35.2	54.9	54.9	1.35	2.01	2.01	1.54	5.37	42.03
Xnio-2.0.0CR2	2.1.0CR1	3	3	26.6	54.9	54.9	1.52	3.01	3.01	2.24	9.84	131.10
Xnio-2.0.0CR2	2.1.0CR1	5	5	17.9	54.9	54.9	1.70	5.02	5.02	4.76	17.65	4804.70
Xnio-2.0.0CR2	2.1.0CR1	7	7	13.4	54.9	N.A.	1.78	7.02	N.A.	8.31	28.58	> 1 h

fied non-standard Java VM [6, 12, 23] that allows loading multiple versions of the same named classes multiple times. However, it does not share common code either.

To share common code, Paal et al. [20] propose a customizable hierarchical class loader approach to separate the component-based application space so that two applications that share the same system class loader share all the code loaded by the system class loader. However, this approach requires manually configuring the hierarchy of class loaders. It also lacks flexibility, allowing component-based applications to communicate with each other using only the types from the core of their system. Such a class loader approach does not separate the Java core library space either [15].

Deduplication is a general approach to address redundancy in memory contents at run-time [26]. Deduplication shares identical memory blocks in virtualized execution environments. This approach is independent of target language and platforms and has recently been extended to sharing contents of similar (but not identical) memory blocks as well [10]. However, it is less specific and less efficient than our code transformation-based approach, and is not amenable to program analysis as it is agnostic of the structure of the underlying data.

Project centralization is a general solution to solve the version conflict problem, which allows sharing common code. Our approach extends existing process centralization [5, 22, 1] for component-based systems with multiple versions, such that both static storage and runtime memory of these systems can be shared. Our approach is also able to separate the Java core library level runtime space for different applications through code instrumentation.

Software merging is another technique related to project centralization. It is widely used in revision control systems [19, 4] to merge files with the same name that have been revised differently. Mens [18] makes a comprehensive survey on the field of software merging techniques and points out that current techniques give no guarantees about the runtime behavior of the programs based on the merged code. On the other hand, project centralization considers sharing common code among different projects without merging files. Our main concern is to share common code of multiple component-based applications in a distributed system, where similar installations are duplicated among many peers. We will consider combining file merging techniques with our project centralization in future work.

6. CONCLUSION AND FUTURE WORK

In this paper, we have investigated project centralization to manage component-based systems with multiple versions. We have formalized the D-graph representation of projects and transformed project centralization into a graph coloring problem. We have presented an algorithm based on existing graph coloring solutions. The experiments on real-world projects demonstrate the effectiveness of our method in sharing common code and resolving version conflicts, showing the usefulness of our coloring based approach in practice.

Future work will use project centralization as a general framework for verifying distributed applications. We also consider using project centralization to manage related software, such as software products from the same software product line. Furthermore, it would be interesting to convert our D-graph representation of a project set into the DIMACS format that can be used by *SAT solvers* to improve the efficiency to calculate the optimal solution.

7. ACKNOWLEDGMENTS

This work was supported by Global COE ⌈Secure-Life Electronics⌋ Program from the University of Tokyo, Japan.

8. REFERENCES

[1] C. Artho and P.-L. Garoche. Accurate centralization for applying model checking on networked applications. In *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pages 177–188, Tokyo, Japan, 2006.

[2] G. Birkhoff. *Lattice theory, the 3rd Edition*. Amer. Math. Soc. (AMS), 1995.

[3] H. L. Bodlaender and D. Kratsch. An exact algorithm for graph coloring with polynomial memory. Technical Report UU-CS-2006-015, Department of Information and Computing Sciences, Utrecht University, 2006.

[4] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, 1998.

[5] G. Czajkowski. Application isolation in the Java virtual machine. In *Proc. 15th Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA 2000)*, pages 354–366, Minneapolis, Minnesota, USA, 2000. ACM.

[6] L. Daynès and G. Czajkowski. Sharing the runtime representation of classes across class loaders. In *Proc. 19th European Conf. on Object-Oriented Programming (ECOOP 2005)*, volume 3586 of *LNCS*, pages 97–120. Springer, 2005.

[7] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system. In *Proc. 10th ACM/IFIP/USENIX Int. Conf. on Middleware*, pages 18:1–18:20, Urbanna, Illinois, USA, 2009.

[8] T. Dumitras, P. Narasimhan, and E. Tilevich. To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains. In *Proc. ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA 2010)*, pages 865–876, Reno Tahoe, Nevada, USA, 2010.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) language specification, the 3rd Edition*. Addison-Wesley Professional, 2005.

[10] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, 2010.

[11] P. Hnetynka and P. Tuma. Fighting class name clashes in Java component systems. In *Modular Programming Languages*, volume 2789 of *LNCS*, pages 106–109. Springer, 2003.

[12] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *Proc. 2013 Int. Conf. on Softw. Eng. (ICSE 2013)*, pages 612–621, San Francisco, CA, USA, 2013.

[13] M. Kubale. *Graph Colorings*. Contemporary mathematics v. 352. Amer. Math. Soc. (AMS), 2004.

[14] M. Lehman and J. Ramil. Software evolution in the age of component-based software engineering. *Softw., IEEE Proc.*, 147(6):249–255, 2000.

[15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification, Java SE 7 Edition*. Addison-Wesley Prof., 2013.

[16] L. Ma, C. Artho, and H. Sato. Analyzing distributed Java applications by automatic centralization. In *Computer Software and Applications Conference Workshops (COMPSACW 2013)*, pages 691–696, Kyoto, Japan, 2013.

[17] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proc. 14th European Conf. on Object-Oriented Programming*, volume 1850 of *LNCS*, pages 337–361. Springer, 2000.

[18] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[19] B. O'Sullivan. Making sense of revision-control systems. *Commun. ACM*, 52(9):56–62, 2009.

[20] S. Paal, R. Kammüller, and B. Freisleben. Customizable deployment, composition, and hosting of distributed Java applications. In *On the Move to Meaningful Internet Systems*, volume 2519 of *LNCS*, pages 845–865. Springer, 2002.

[21] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[22] S. D. Stoller and Y. A. Liu. Transformations for model checking distributed Java programs. In *Proc. 8th Int. SPIN Workshop on Model checking of Software*, pages 192–199, Toronto, Ontario, Canada, 2001.

[23] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a VM-centric approach. In *Proc. 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2009)*, pages 1–12, Dublin, Ireland, 2009.

[24] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[25] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[26] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.