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, 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. Dumitra¸ s 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 summarizes our previous approach [16]. Section 3 elaborates our D-graph representa-
tion 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 Project1 because class C references A. In Fig. 1(a), Project1 and Project2 can share most of their classes except for C, where different versions are used. Compared to Project2, Project3 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. Project1 renames its classes to P1.C, P1.B, and P1.Main to separate the version space. Similarly, Project2 and Project3 share classes C and B, and Project2 renames its class Main to P2.Main. Classes Main and Unique in Project3 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 name and implementation, respectively. Given two classes cl1 and cl2, cl1 and cl2 are equivalent, denoted by cl1 = cl2, if they form a Type-1 clone pair [21], where cl1.name is identical to cl2.name, and cl1.code and cl2.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 NAME(p) = {cl.name | cl \( \in \) p} as the set containing all the class names in p, and P \( \cap \) cln = \{p \( \in \) P | cln \( \in \) NAME(p)\} as the set of all projects that contain the class named cln.

Definition 3. Let p be a project, and cln1 and cln2 be two class names. Project renaming substitution p[cln1/cln2] is defined as a project in which p substitutes its class name cln1 for cln2. Substitution includes class names and references to them. A renaming substitution p[cln1/cln2] is a normal substitution if cln1 \( \notin \) NAME(p) and cln2 \( \in \) NAME(p).

Definition 4. Let p1 and p2 be two projects; p1 is equivalent to p2, denoted by p1 = p2, 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 pcentr such that \( \forall p \in P, \exists p' \subseteq pcentr, p = p' \). 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 $cl_1 \cdot code$ references $cl_2 \cdot name$.

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.** \(unique(p, P) = \{ cl \in p \exists q \in P \setminus p. cl \cdot name \notin NAME(q) \}.\) A unique class has a unique name across all projects in $P$.

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

3. **Shared Class.** \(shared(p, P) = \{ cl \in p \exists q \in P. cl \cdot name \in \{ NAME(p) \cap NAME(q) \} \wedge p \cdot cl = GetClass(q, cl \cdot name) \}.\) 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 Project1, but it is both shared and a conflict class in Project2 and Project3. Similarly, class $Main$ is a conflict class in Project3, and it is both shared and a conflict class in Project1 and Project2.

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

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 $\forall p \in P. conflict(p, P) = \emptyset$, the optimal result is the union of all projects in $P$. $\bigcup_{p \in P} p$. If conflicts are present, $\exists p \in P. conflict(p, P) \neq \emptyset$, 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 Project1 and Project2 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 $m$ classes 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 $\mathcal{O}(m^2 \cdot n)$ 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 $P = \{ p_1, p_2, p_3, p_4 \}$ 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 $(P \uparrow cln, CE)$, such that if there exist two projects $p, p' \in P \uparrow cln$ and $(p, p') \in CE$, and $p'$ cannot share the class named $cln$.

Each node in a constraint graph of name $cln$ is a project containing a class name $cln$. 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 $G = (P \uparrow cln, CE)$ be a constraint graph of $cln$ in $P$, and $P'$ be a project set. We write the subgraph of $G$ to $P'$ as $(P', CE')$ (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 $CG$.

Definition 9. We define $\sqsubseteq_{cg} \in CG \times CG$ as a binary relation such that for any two constraint graphs $G_1, G_2 \in CG$ 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 $G_{1 \sqcup 2} = \langle P_1 \cup P_2, CE_1 \cup CE_2 \rangle$.

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

Let $cln$ be a class name in $P$, we write $CG_{cln}$ as the sub-domain of $CG$, where $CG_{cln} \subseteq CG$ and $\forall G \in CG_{cln}. G = \langle P', CE' \rangle \Rightarrow P' = P \cap cln$. It is not difficult to prove that the partially ordered set $(CG_{cln}, \sqsubseteq_{cg}, CG_{cln}, \sqsubseteq_{cg} \cap \sqcup_{cln})$ is a complete lattice [2] with, $\forall X \subseteq G, X = \{ x_1, x_2, \ldots, x_n \}$:

- a least upper bound $\sqcup_{cln} X = x_1 \sqcup_{cg} x_2 \sqcup_{cg} \ldots \sqcup_{cg} x_n$,
- a greatest lower bound

$\sqcap_{cln} X = \{ y \mid \forall x \in X, y \sqsubseteq_{cg} x \}$,
• a least element $\perp_{cg} = \langle P \uparrow cln, \emptyset \rangle$,
• a greatest element $\top_{cg} = \langle P \uparrow cln, CE_{\text{greatest}} \rangle$, where $CE_{\text{greatest}} = \{(m, n) | n, m \in P \uparrow cln \land m \neq n \}$.

We extend the constraint graph to a constraint lattice.

**Definition 10.** A constraint structure $CS$ in a project set $P$ consists of a name in $\text{NAMES}(P)$ (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 of 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 $e.set = \{p \in (P \uparrow l.name \land P \uparrow m.name) \mid \text{GetClass}(p, l.name) \to \text{GetClass}(p, m.name)\}$ such that:

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

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 $\text{Pred}(n) = \{m \mid (m, n) \in E\}$. 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](image)

**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 $\nu(G) = \langle |N|, E \rangle$, where $|N|$ represents the nodes of $G$ that ignore all their constraint graphs.

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

**Definition 14.** Let $\mathcal{G}$ 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 $\mathcal{G}$. $G$ and $G'$ have the binary relation $G \sqsubseteq G'$ if and only if $\forall n \in N, n.name = n'.name \Rightarrow n \sqsubseteq_{cs} n'$. The least upper bound of $G$ and $G'$ is $G \sqcup G' = \langle N'', E \rangle$, where $N'' = \{m \sqcup_{cs} n | n \in N \land m \in N' \land m.name = n.name\}$.

Assuming $\text{NAMES}(P) = \{cln_1, cln_2, \ldots, cln_k\}$, the constraint graph domain and underlying graph of $P$ be $\mathcal{G}$ and $|\mathcal{G}| = \langle |N|, \nu(G) \rangle$, respectively. It is not difficult to prove that $(\mathcal{G}, \sqsubseteq, \sqcup, \sqcap, \top, \perp)$ is a complete lattice [2] such that $\forall X \subseteq \mathcal{G}$ 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 $\sqcap_X = \bigcup\{y \mid y \sqsubseteq x \}$,
- a least element $\perp = \langle \{cln_1, \perp\}, \cdots, \{cln_k, \perp\} \rangle, E\rangle$,
- a greatest element $\top = \langle \{cln_1, \top\}, \cdots, \{cln_k, \top\} \rangle, E\rangle$.

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 $G = (N, E)$ be a D-graph of $P$. Let $e \in E$ be an edge and $e = (m, n)$, where $m.CG = \{P\ name, CE\}$ and $n.CG = \{P\ name, CE\}$. The edge $e$ is a conflict edge if $3p, q \in e.set, (p, q) \in CE$.

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 $B$ to $C$ in Fig. 2(a) is a conflict edge. $P_1$ and $P_2$ must hold a different version of class $A$, 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 $G = (N, E)$ 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 $IN_0(n)$ and $OUT_0(n)$ are the corresponding initial conditions, respectively. The functions in equation 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].

$$\begin{align*}
IN(n) &= \bigcup_{m \in Pred(n)} OUT(m) \cup (m,n).set \\
OUT(n) &= IN(n) \cup OUT(n) \\
IN_0(n) &= \emptyset \\
OUT_0(n) &= n.CG
\end{align*}$$

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).

### 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 fix 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 (lines 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 $G = (N, E)$ with $\#N = m$ and $\#E = l$.
Algorithm 2 Solve Constraint Equations

```
1: function EQUATION_SOLVER
   ▷ 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 ∈ TopoSCCs do
   ▷ Visit each SCC in topological order
3:   repeat
4:      ▷ Repeat if constraints of a node in SCC change
5:       for n ∈ graph.nodeSet ∧ n ∈ SCC do
6:          IN(n) ← (\{m | \exists pred(n) in OUT(m), m \} ∪ \{m, n\}).set
7:          OUT(n) ← IN(n) ∪ OUT(n)
8:          n.CG ← OUT(n)
9:       end for
10:   until ∀n ∈ (graph.nodeSet ∩ SCC).IN(n) and OUT(n) do not change
11: end for
12: return graph
```

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 \( \log \) steps. 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 \( \log \) steps. The overall experimental result is summarized in Table 2. We have five experimental settings for 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 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.

Algorithm 1

```
1: function FINDING_SOLVER
   ▷ Finding the minimal D-graph so that the number of classes in the output is minimal
Input: graph: the D-graph
Output: graph: the minimal valid D-graph
2:   for SCC ∈ TopoSCCs do
3:      ▷ Visit each SCC in topological order
4:         repeat
5:            ▷ Repeat if constraints of a node in SCC change
6:               for n ∈ graph.nodeSet ∧ n ∈ SCC do
7:                  IN(n) ← \{m | \exists pred(n) in OUT(m), m \} ∪ \{m, n\}.set
8:                  OUT(n) ← IN(n) ∪ OUT(n)
9:                  n.CG ← OUT(n)
10:             end for
11:         until ∀n ∈ (graph.nodeSet ∩ SCC).IN(n) and OUT(n) do not change
12:   return graph
```

The optimal solution requires \( O(n \cdot m) \) run time. The complexity analysis in Section 3.3 shows that our solution is linear in 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.

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 Benchmarks

<table>
<thead>
<tr>
<th>Project name / version</th>
<th>Edit Hp</th>
<th>Ganymed-s2</th>
<th>Jsmpp</th>
<th>Kryonet</th>
<th>Mime4j-core</th>
<th>Anio</th>
<th>Netx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Build</td>
<td>Build</td>
<td>Build</td>
<td>Build</td>
<td>Build</td>
<td>Build</td>
<td>Build</td>
</tr>
<tr>
<td></td>
<td>2.4.0</td>
<td>2.0</td>
<td>0.7.2</td>
<td>2.1.0</td>
<td>2.0.0CR1</td>
<td>1.0</td>
<td>0.4</td>
</tr>
<tr>
<td>#Cl. (*.class)</td>
<td>156</td>
<td>133</td>
<td>201</td>
<td>79</td>
<td>104</td>
<td>61</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 2: Experimental Results of Project Centralization

<table>
<thead>
<tr>
<th>Project name / version</th>
<th>Inst.</th>
<th>Shared [%]</th>
<th>Storage Ratio</th>
<th>Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Simple</td>
<td>Greedy</td>
<td>Optimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 share 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


