Managing Product Variants by Project Centralization

Lei Ma, Cyrille Artho, and Hiroyuki Sato

***Abstract*—Many software systems are evolving continuously. The changes are reflected in a sequence of revised products, including both version updates and related product variants that are created by the clone-and-own approach. Separate management and analysis of these product variants waste storage and hinders uncovering their commonalities and variations for maintenance. In this paper, we explore the project centralization approach to manage product variants. Our technique shares common code whenever possible while keeping the version space of each project separate. We present the key issues of project centralization and its algorithms. We perform several case studies, where project centralization is applied to real-world software projects, demonstrating the potential usefulness of our approach.**

***Index Terms*—Software evolution, software management, project centralization, multiple versions, program analysis.**

I. Introduction

A software system continuously changes to increase its functionality, fix bugs, and adapt to new requirements over its life cycle [1]. Such changes are released as a sequence of updated revisions. Some related product variants are also created by coping and modifying existing ones (the clone-and-own approach), which is a common practice for developing new software products [2]. As many similar product variants are developed, management and analysis of these products effectively becomes very important.

Managing each of these products separately wastes storage by code duplication and causes redundancies for analysis and verification. For example, if a function under test exists in multiple products with both the same implementation and runtime behavior, independently exercising the same test case for such a function for each product causes redundant executions [3]. If multiple variants of a software system interact in a distributed system, analysis and verification of these applications also create a challenge in representing these multiple versions for analysis tools and execution platforms like Java and C#, which are designed to handle only single version [4] .

Several strategies have been proposed to manage multiple variants of a software system during the past decades. Some advocates refactor them into Software Product Lines (SPL) [5]-[7], and many others manages them in a revision control system [8]-[10]. A difficulty of refactoring multiple similar product variants into a SPL is to extract the commonality and variability that is usually represented as a feature model. However, this needs domain analysis to identify features and establish the connections between a feature and its corresponding code, which proves to be difficult to automate and lacks accuracy [11], [2]. On the other hand, as a main challenge of software management using a revision control system, we must resolve version conflicts when merging existing products [8]. A version control system usually adopts a text-based comparison to track changes so that different types of documents can be handled. However, the unawareness of underlying language structure hinders further analysis for the multiple product variants by existing tools. Although some language structure-aware merging strategies have been developed to handle different languages, these techniques do not guarantee to preserve the behavior of each product [9], [10].

In our previous work [4], we have proposed project centralization to resolve version conflicts of multiple product variants in a distributed system. It enables the analysis and verification of the system as a whole by existing tools like Java PathFinder [12]. Project centralization represents the code repositories of all product variants using a single centralized repository while preserving the behavior of each original.

In this paper, we report our preliminary study to explore the usefulness of project centralization for managing multiple product variants to avoid code duplication while preserving the behavior of each product. We perform case studies on three real-world Java applications, containing 10 variants for each application with over two million lines of code in total, to demonstrate the feasibility of project centralization for managing similar product variants and to point out directions for future research. While our implementation supports Java bytecode, the concepts presented in this paper generalize to other managed programming languages and runtime platforms.

The rest of this paper is organized as follows: Section II first illustrates the basic concepts of project centralization. Then, we summarize two project centralization algorithms published earlier [4]. Section III reports on our case studies where we apply our tool to real-world nontrivial projects. After discussing related work in Section IV, Section V concludes and presents future work.

II. Project Centralization

In this section, we present the concepts related to project centralization. We consider sharing common code on a class file level (without merging files).

A. Concepts and Definitions

A Java class is uniquely identified by its name and implementation. For a class *cl*, we use *cl.name* and *cl.code* to

Manuscript received September 25, 2013; revised November 26, 2013. This work was supported in part by Global COE Secure-Life Electronics Program from the University of Tokyo, Japan.

Lei Ma and Hiroyuki Sato are with the Department of Electrical Engineering, the University of Tokyo, Japan (e-mail: malei@satolab.itc.u-tokyo.ac.jp, schuko@satolab.itc.u-tokyo.ac.jp).

Cyrille Artho is with the Research Institute for Secure Systems, AIST, Japan (e-mail: c.artho@aist.co.jp).

DOI: 10.7763/LNSE.2014.V2.122

denote its name and implementation, respectively. Given two classes cl_1 and cl_2, we say they are equivalent, denoted by $cl_1 = cl_2$, if they form a **Type-1** clone pair [13], where $cl_1.name$ and $cl_2.name$ are identical, and $cl_1.code$ and $cl_2.code$ are also identical except for variations in whitespace, layout and comments.

A *project* is a set of classes, where no two classes in a project share the same name. Let p be a project. We denote the number classes of p by $\#p$. We write NAME(p) as the set of all class names in p. A project presents an abstract view of the class repository of a software product.

Definition 1. Let cln_1 and cln_2 be two class names. Project *renaming substitution* $p[cln_1/cln_2]$ is defined as a project, where cln_1 is substituted for cln_2 in p. A renaming substitution for p is a *normal substitution* if $cl_1 \notin$ NAME(p) and $cl_2 \in$ NAME(p).

Definition 2. Let p_1 and p_2 be two projects. The behaviors of p_1 and p_2 are equivalent, denoted by $p_1 = p_2$, if they can be renamed to an identical project by normal renaming substitutions.

Definition 3. *Project centralization* of a project set P transforms P into a 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 CENTRA(P).

Project centralization represents multiple projects by one centralized project, entailing each original project preserves its behavior after centralization. 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 $cl_2.name$. If class cl_1 and 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 behavior. Let p_1 and p_2 be two projects. The behaviors of p_1 and p_2 are equivalent if they can be renamed to an identical project by normal renaming substitutions.

We use the A.D. value to measure the average complexity of the reference dependencies for all projects in a project set.

Definition 4. Let P be a project set. The Average Dependency (A.D.) of P is defined as:

$$A.D. = \frac{\sum_{p \in P} \#\{(cl_1, cl_2) \mid cl_1, cl_2 \in p \wedge cl_1 \rightarrow cl_2\}}{\#P} \quad (1)$$

Let P be a set of projects. To perform project centralization correctly, we classify the classes of a project $p \in P$ into the following categories:

- *Unique class*. A unique class of a project $p \in P$ has a unique name in p, which does not occur in the other projects.
- *Conflict class*. The name of a conflict class in $p \in P$ also exists in another project $p' \in P$ but with a different implementation.
- *Shared class*. A shared class of p shares both its name and implementation with some class in another project.

Consider the general scenario of centralizing projects from a project set P. There may exist multiple solutions that satisfy Definition 3. If no conflict classes exist in any project of P, the project centralization can directly take the union of all projects in P as the centralized result. If some project in P contains a conflict class, it needs to be separated while trying to retain shared classes as much as possible.

B. Example

Fig. 1 shows an example for centralizing three projects. The edges in a project represent the class dependencies. For example, the edge from class A to B in Project$_1$ shows that they have the dependency relation $A \rightarrow B$. Project$_1$ and Project$_2$ share most of the classes except for C where different versions are used. Compared with Project$_2$, Project$_3$ has a different version of *Main* and a new class *Unique*. In this example, class A and B are shared classes in all projects. The cases for 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* in Project$_3$ is a conflict class, and it is both a shared and conflict class in Project$_1$ and Project$_2$.

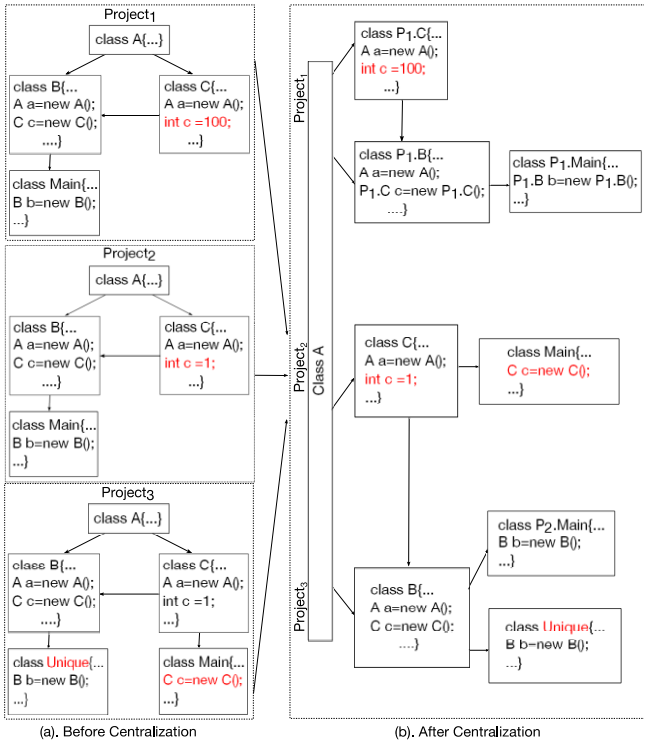

Fig. 1. Project centralization example.

C. Simple Algorithm and Its Limitations

The main issue of project centralization is to properly separate the class versions 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 to be shareable anymore, as their internal references to other classes are renamed differently across projects. Consider the example in Fig. 1 before centralization. Project$_1$ and Project$_2$ can share class B before project centralization. They have to rename their class C to a different name to resolve 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.

Given a project set of n size, our simple algorithm [4] renames all the conflict classes of the first n -1 projects and propagates the renaming effect by traversing class dependency relations of each project. It does not distinguish between a conflict class and a class that is both shared and

conflict. It renames the class as long as it is a conflict class and the centralized result depends on the order of the projects. For example, when centralizing projects Fig. 1 in the order of $Project_2$, $Project_1$, and $Project_3$, the simple algorithm cannot share classes *B* and *C* in the centralized result. However, a better solution after centralization is shown in Fig. 1, where classes *B* and *C* are both shared by $Project_2$ and $Project_3$.

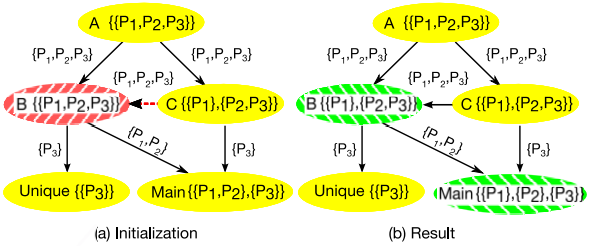

Fig. 2. P-graph representation of a project set.

D. Graph Based Approach

In this section, we summarize our optimized project centralization solution [4] based on a graph representation for all projects. We define the *P-graph* representation for a project set and discuss the optimized project centralization algorithm. In the rest of this section, we fix *P* as an arbitrary project set for centralization.

For a class name *cln* in *P*, we use a partition of projects to represent the version relations of classes that share that name. A project partition of *cln* is a set of projects such that two projects occur in the same subset of the partition if they share the same version of the class named *cln*.

Definition 5. Let *cln* be a class name in *P*. A *partition structure* of *cln* is the pair of *cln* and a partition of *cln*. Let *PS* be a partition structure, we denote its name and partition by *PS.name* and *PS.partition*, respectively.

Definition 6. A *P-graph* <*N*, *E*> of *P* consists a set of partition structures as a set of nodes, and an edge set *E* to represent the dependency of these constraint structures, which satisfies: 1) For each class name *cln* in *P*, there exists exactly one partition structure node n$\in N$ with *n.name* = *cln*. 2) For two nodes *m*, *n* $\in N$, the existence of an edge *e* from *m* to *n*, denoted by (*m*, *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 $P\uparrow n.name$. Each edge *e* $\in E$ is associated with a set of projects, denoted by *e.set*, that satisfy such a condition.

Let *G*=<*N*, *E*> be a P-graph. Each node *n* $\in N$ is a partition structure <*n.name*, *n.partition*> that represents all the version relations of the class named *n.name* in *P*. For each edge *e* $\in E$, its set *e.set* contains at least one project.

Fig. 2 shows the corresponding initialized P-graphs of project set in Fig. 1. The larger node is the partition structure node, in which its name and partition of the class 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 subset, meaning that all these projects have the same version of class named *A*. The label of an edge in a P-graph shows those projects, in which two partition structure nodes connected by that edge have a dependency relation. For example, the edge from node *B* to *Main* indicates that the classes named *B* and *Main* have a dependency relation in both P_1, P_2, but not in P_3.

TABLE I: OVERVIEW OF ARGOUML PRODUCT VARIANTS

Pr.	DrJava	LOC	Size (KB)	#Class
1	Release 20130901-r5756	89793	11895	3936
2	Release 20120818-r5686	89786	11895	3925
3	Release 20110822-r5448	89736	11886	3925
4	Release 20100913-r5387	88401	11659	3877
5	Release 20100816-r5366	88275	11655	3877
6	Release 20100711-r5314	87556	11572	3837
7	Release 20100507-r5246	87258	11499	3824
8	Release 20100415-r5220	85622	11506	3792
9	Release 20090821-r5004	81239	9798	3251
10	Release 20090803-r4975	81044	9711	3242
Pr.	ArgoUML	LOC	Size (KB)	#Class
1	All optional feature enabled	120348	5147	1915
2	All optional feature disabled	82924	3669	1494
3	Only Logging disabled	118190	5018	1915
4	Only Cognitive disabled	104209	4431	1678
5	Only Sequence Diagram disabled	114969	5033	1881
6	Only Use Case Diagram disabled	117636	5032	1874
7	Only Deployment Diagram disabled	117201	5024	1882
8	Only Collaboration Diagram disabled	118769	5086	1896
9	Only state Diagram disabled	116431	5008	1880
10	Only Activity Diagram disabled	118066	5036	1897
Pr.	HealthWatcher	LOC	Size (KB)	#Class
1	Base–no extensions applied	5228	261	90
2	Command pattern applied	5646	273	94
3	State pattern applied	6112	302	106
4	Observer pattern applied	6222	309	108
5	Adapter pattern applied	6379	314	110
6	Abstract Factory pattern applied V_1	6417	318	114
7	Adapter pattern applied	6441	319	118
8	Abstract Factory pattern applied V_2	6468	321	122
9	Evolution–new functionality added	7709	389	134
10	Exception Handling applied	7591	389	137

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

Definition 7. Let *G*=<*N*, *E*> be a P-graph of *P* and *e* $\in E$ be an edge, where *e* = (*m*, *n*) with *m*, *n* $\in N$. The edge *e* is a conflict edge if there exists two projects p_1, $p_2 \in e.set$ so that p_1 and p_2 occur in the same subset in *n.partition* but not in *m.partition*.

A conflict edge captures cases where classes must be renamed to separate their version space. This ensures that project centralization preserves the behavior of each project. For example, the dashed edge in Fig. 2 is a conflict edge because P_1 and P_2 exist in the same subset of *B.partition* but not in *C.partition*. Let *e* = (*m*, *n*) be a conflict edge with *m*, *n* $\in N$. The conflict can be resolved by refining *n.partition*. After a conflict is resolved, it may introduce more conflict edges. A correct separation of the version space of each project ensures that no unresolved conflicts remain.

Our optimized algorithm [4] adopts the P-graph representation for projects. It takes a project set as input and outputs a centralized project that satisfies Def. 3. The

algorithm first initializes the P-graph of *P* by collecting class names, creates a constraint structure node for each of them, and adds their corresponding dependency edges. Then, the conflict edges are resolved in a topological order of the P-graph by refining partitions. The algorithm iteratively resolves all these conflict edges. For example, Fig. 2 gives a resolved P-graph result with no conflict edges. After resolving all conflict edges, the last step performs normal renaming substitution according to the obtained P-graph so that two projects that occur in the same subsets of a partition share that class. After performing such a normal substitution according to Fig. 2, we can obtain the same project centralization results in Fig. 1.

III. Empirical Study

In this section, we present the case studies we have conducted to investigate the feasibility of managing product variants by project centralization. We have implemented the project centralization solutions in a tool that performs bytecode transformation using the ASM bytecode library [14]. It takes a project set as input and outputs the centralized projects. In the rest of this section, we first define the evaluation criteria for measurement. Then, we explain our case studies and discuss the results in detail.

A. Evaluation Criteria

We identify three criteria to evaluate the effectiveness in sharing code and the run time of a centralization algorithm.

Criterion 1. Effectiveness of sharing common code is defined as Sharing Factor (S.F.), which is the ratio of, shared classes to output classes:

$$S.F. = \frac{\#ClassShared}{\#OutputClass} \quad (2)$$

A class in the output is counted as shared if at least two projects share it after centralization. Consider the project centralization results in Fig. 1. Classes A, B, C are shared by multiple projects in the output classes, but classes Unique and Main are not. S.F. ranges from 0 to 1; the larger its value, the more classes are shared. The trivial renaming approach renames all classes of each project and shares no classes; its value of S.F. is therefore 0.

Criterion 2. The effectiveness of saving storage is defined as the ratio of storage usage after centralization to the one before centralization.

This measure shows the storage that can be saved by project centralization. It also indicates the possible runtime memory that can be saved when simulating an application by process centralization [4], where a class is only necessary to be loaded to the JVM once by centralization.

Criterion 3. Performance is the execution time of project centralization algorithm and total time that includes additional time to write transformed results to disk, but not including the prepossessing and initializing time to build internal data structure from class files from disk.

All experiments were measured 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.

B. Case Studies and Results

The overview of product variants in our case studies is summarized in Table I, which contains both version variants and product variants from a SPL. Column one lists the product number and column two gives brief descriptions of product variants for each software system by enabling and disabling some features. Column three shows the lines of source code (without comments and whitespaces) for each product. Column four and five list the product size and number of classes (*.class files), respectively.

DrJava [15] in our first case study is a lightweight development environment for writing Java programs. We use its most recent ten version variants released in the last four years. In the second and third case studies, we use the product variants generated from existing software product line ArgoUML-SPL [16], [17] and Health-Watcher [18], [19], respectively. ArgoUML-SPL is the software product line for the UML modeling tool ArgoUML, and HealthWatcher is a cloud computing application that manages health records and complaints.

Experimental results, where project centralization algorithms are applied to all the selected products of a software project, are summarized in Table II. For each input project set, we give its total number of classes (*.class files), storage usage, and average dependency in columns three, four and five, respectively. We compare the simple algorithm and optimized algorithm in the number output classes, S.F., storage saving ratio, and algorithm execution time and total time. We show these results in columns OUTPUT #Class, S.F., Storage, Alg.Time and Total Time, respectively.

According to the input metrics, DrJava and ArgoUML are both large-size projects, and HealthWatcher is smaller. Although the number of input classes for DrJava is twice as large as in ArgoUML, the average dependency for classes in DrJava is much larger.

In all experimental settings, the optimized algorithm performs better by outputting fewer classes, sharing more common code and saving more storage. Compared with the simple algorithm, the optimized algorithm shares 21.9%, 22.9% , and 35.2% more classes as indicated by the column S.F. The optimized algorithm also saves 22.0%, 12.9% and 29.2% more storage indicated by column Storage. However, the algorithm execution time and total time of the simple algorithm is faster. As the optimized algorithm needs to resolve conflict edges iteratively in the P-graph, the execution time strongly depends on the average dependency between input project sets. Its execution time becomes slow when the number of input classes and average dependency in the input project set are large, like for DrJava. From these case studies, we find that our optimized project centralization solution is efficient enough to handle large project sets, and effective in sharing common code and saving storage. The centralized project preserves the behavior of all original projects, which can be directly analyzed and verified to reveal defect of original projects.

C. Threat to Validity

The main threat to our work is that we only performed case studies on three open source projects and selected ten variants for each project. However, the selected projects are

real-world non-trivial projects and have been also be used in case studies by many other researchers. Another threat is that our approach makes project centralization on the file level without merging class files. It cannot share the classes that are slightly different. This threat can be diminished by applying the same methodology of project centralization to a finer level such as the method and attribute levels. We leave such a refinement as future work.

TABLE II: EXPERIMENTAL RESULTS OF PROJECT CENTRALIZATION

Project	Algorithm	INPUT			OUTPUT #Class	S.F. (%)	Storage (%)	Alg. Time (ms)	Total Time (ms)
		#Class	Size (KB)	A.D.					
DrJava	Simple	37486	113076	17199	29194	6.7	82.2	6920	10949
	Optimized				22247	28.6	60.2	37166	40063
ArgoUML	Simple	18312	48484	3121	6399	27.7	43.0	2273	3286
	Optimized				4485	50.6	30.1	6959	7706
HealthWatcher	Simple	1133	3195	373	771	10.9	82.1	1128	1242
	Optimized				448	46.1	53.8	1513	1585

IV. RELATED WORK

Hnetynka *et al*. [20] originally discussed the version conflict problem in component-based systems written in Java. They adopt the renaming approach by augmenting the class name of each variant with a version identifier during dynamic class loading, which does not share common code.

Managing the product variants by revision control tools is commonly used in practice. A difficult issue is to represent multiple simultaneous changes of a single product. Software merging is a widely used technique similar to project centralization, which merges multiple product variants into single one. However, Mens [8] points out that current software merging techniques give no guarantees about the behavior of merged code. On the other hand, project centralization preserves the behavior of each project so that program analysis techniques can be directly applied to the centralized project as shown in our previous work [4].

Managing product variants to refactor them into a SPL is another technique related to project centralization [11]. However, such techniques require manual analysis based on requirement to identify features and establish traces between features and their location in the code. Existing approaches are mostly work on the model level [2], [11], [7]. Our project centralization focuses on sharing common code of existing products to reduce maintenance cost and remove version conflicts for program analysis.

V. CONCLUSION AND FUTURE WORK

In this paper, we explore the usefulness of project centralization for managing similar product variants. We first present the basic concepts and definitions of project centralization. Then, we discuss a simple solution and an optimized one. We conduct empirical evaluations of our approaches on three real-world large software systems. The result demonstrates the effectiveness of our technique in managing similar product variant by avoiding redundant code. Based on this exploratory work, our future work will refine project centralization to method level and compare its effectiveness with the current approaches in sharing common code of multiple product variants.

REFERENCES

[1] M. M. Lehman and J. F. Ramil, "Software evolution in the age of component-based software engineering," in *Proc. IEE Softw.*, vol. 147, Dec. 2000, pp. 249–255.

[2] J. Rubin, K. Czarnecki, and M. Chechik, "Managing cloned variants: a framework and experience," in *Proc. 17th Int. Softw. Product Line Conf*., 2013, pp. 101–110.

[3] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *Proc. 23rd IEEE/ACM Int. Conf. on Autom. Softw. Eng*., 2008, pp. 218–227.

[4] L. Ma, C. Artho, and H. Sato, "Improving automatic centralization by version separation," *IPSJ Trans. on Programming*, 2013.

[5] D. Faust and C. Verhoef, "Software product line migration and deployment," *J. of Softw. Practice and Experience*, vol. 33, pp. 933–955, 2003.

[6] T. Mende, R. Koschke, and F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," *J. Softw. Maint. Evol.*, vol. 21, no. 2, pp. 143–169, 2009.

[7] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Proc. 33rd Int. Conf. on Softw. Eng*., 2011, pp. 461–470.

[8] T. Mens, "A state-of-the-art survey on software merging," *IEEE Trans. Softw. Eng*., no. 5, pp. 449–462, 2002.

[9] S. Apel, O. Lessenich, and C. Lengauer, "Structured merge with auto-tuning: balancing precision and performance," in *Proc. 27th IEEE/ACM Int. Conf. on Autom. Softw. Eng*., 2012, pp. 120–129.

[10] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: rethinking merge in revision control systems," in *Proc. 19th ACM SIGSOFT symp. and the 13th Euro. Conf. on Foundations of Softw. Eng*., 2011, pp. 190–200.

[11] J. Rubin and M. Chechik, "A framework for managing cloned product variants," in *Proc. 35th 2013 Int. Conf. on Softw. Eng*., 2013, pp. 1233–1236.

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

[13] 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*., vol. 74, pp. 470–495, 2009.

[14] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A code manipulation tool to implement adaptable systems," *Adaptable and Extensible Component Systems*, 2002

[15] DrJava Project. [Online]. Available: http://www.drjava.org/

[16] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: a case study using conditional compilation," in *Proc. 15th Euro. Conf. on Softw. Maint. and Reeng*., 2011, pp. 191–200.

[17] ArgoUML-SPL. (Sep. 16, 2013). [Online]. Available: http://argouml-spl.tigris.org/

[18] E. Cavalcante, A. Almeida, T. Batista, N. Cacho, F. Lopes, F. C. Delicato, T. Sena, and P. F. Pires, "Exploiting software product lines to

develop cloud computing applications," in *Proc. 16th Int. Softw. Product Line Conf.*, vol. 2, 2012, pp. 179–187.
[19] Health Wactcher. (Sep. 19, 2013). [Online]. Available: http://www.comp.lancs.ac.uk/~greenwop/tao/java.htm
[20] P. Hnetynka and P. Tuma, "Fighting class name clashes in Java component systems," *Modular Programming Languages*, vol. 2789, pp. 106–109, 2003.

Lei Ma is a Ph.D. candidate at Department of Electrical Engineering, the University of Tokyo. He received his B.S. from Shanghai Jiao Tong University in the Department of Computer Science in 2009, and M.S. from the University of Tokyo in Department of Electrical Engineering and Information System. His interests cover various topics related to programming languages, type system, and software management.

Cyrille Artho is a senior researcher at AIST, Japan. 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. After graduation he worked at NII, Tokyo, for two years, and then moved to AIST.

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.