Algebraic and Cost-based Optimization of Refactoring Sequences*

Martin Kuhlemann[1], Liang Liang[2], and Gunter Saake[3]

[1] University of Magdeburg, Germany
martin.kuhlemann@ovgu.de
[2] University of Magdeburg, Germany
leon.liangliang@hotmail.com
[3] University of Magdeburg, Germany
gunter.saake@ovgu.de

Abstract. Software product lines comprise techniques to tailor a program by selecting features. One approach to implement product lines is to translate selected features into sequenced program transformations which extend a base program. However, a sequence translated from the user selection can be inefficient to execute. In this paper, we show how we optimize sequences of refactoring transformations to reduce the composition time for product line programs.

1 Introduction

A feature is a characteristic of a program which is of interest to a user [14]. *Software product lines (SPLs)* comprise techniques to tailor the set of features of a program to user needs [17]. One technique to implement an SPL is to define code transformations which successively apply to a base program and add the desired program characteristics to it. These SPL transformations include aspects [32], refinements [4], refactorings [18], and others.

In SPLs, feature-adding code transformations are abstract operations which a user selects without knowing their implementation. As a result the user (unknowingly) may select transformations that undo each other or that override another transformation's effects. Such a non-optimal *transformation plan* may be selected by accident but may also be meaningful to reuse transformations.[1] While the composition result is correct, the composition process is more expensive than necessary. On the one hand, end-user satisfaction may be increased due

* This paper summarizes and extends the Master's Thesis of Liang Liang [21]. An extended version of this paper with more technical details has been published as a technical report [20]. The authors thank Don Batory and Andreas Lübcke for helpful discussions and for giving hints on earlier versions of this paper. The authors thank the anonymous reviewers of MDPLE 2010 for additional hints.

[1] Suppose in one configuration of an SPL, classes List and ArrayList should switch names then one of them must be renamed twice, e.g., List ↦ TestList ↦ ArrayList. In a second configuration, in which only List exists, List should be renamed into ArrayList, too, and for that both prior refactorings get reused.

to a reduced program composition time. On the other hand, developers benefit when they must compose a number of SPL programs, e.g., during testing.

In this paper, we lean on database optimization techniques and optimize sequences of refactorings translated from a user selection of features. We discuss the theoretical basics as well as our prototype. In case studies we observed that with our prototype we reduced composition time by up to 81%.

2 Background

We introduce the concepts of refactoring along with transformation-based SPLs as these concepts are issue to optimization, later.

2.1 Refactorings

Refactorings are code transformations that alter the structure of code but do not alter its functionality [26]. Refactoring descriptions, like Rename Class, are templates and so a developer has to provide parameters to make the templates executable [25]. For example, to execute a Rename Class refactoring, two parameters must be defined: the class to rename and the new class name.

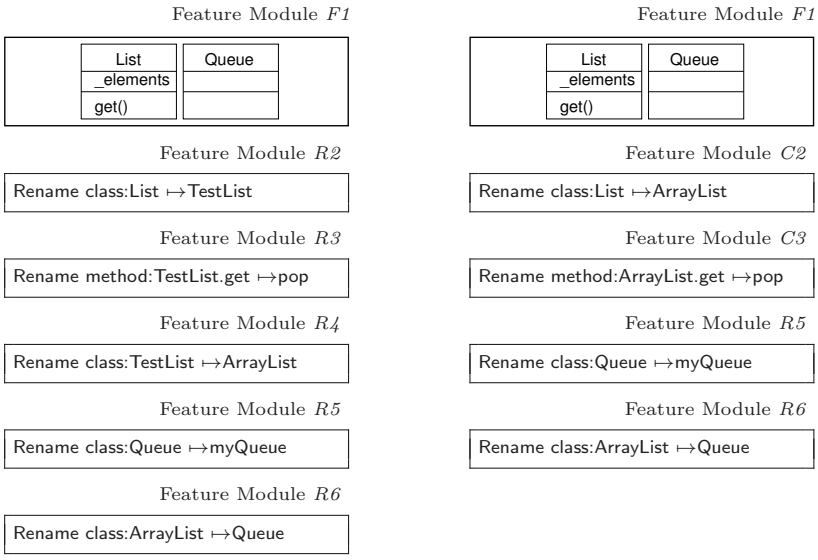
When a refactoring is parameterized and executed, the refactoring engine commonly executes two phases. In the *verification phase*, preconditions are checked in the code to refactor to ensure its transformation does not alter its functionality. For any Rename Class refactoring, the refactoring engine will check whether the class to rename does exist and whether the class created by the refactoring does not exist [28].

In the *transformation phase*, transformations are performed on the code elements specified as parameters for the refactoring. That is, for Rename Class, the specified class is renamed and every reference to the class is updated [10]. In the following, we denote a refactoring R, that replaces a code element X by a code element Y, with $R_{\mathsf{X}\mapsto\mathsf{Y}}$.

2.2 Transformation-Based Software Product Lines

Features are user-visible program characteristics of an SPL and are selected to tailor a program of that SPL [14]. Features can be implemented by code transformations defined in feature modules [4]. In SPLs, feature modules are hidden from the user – the user makes a selection of feature names/descriptions which is translated into a sequence of program transformations. The program transformations make the generated program expose the required features.

Program transformations in SPLs can implement refactorings in order to integrate programs, foster reuse, and to tailor non-functional properties of programs [18,29]. Feature modules which host such refactoring transformations are called *refactoring feature modules (RFMs)* [18]. When a user selects features, that translate to refactorings the structure of the synthesized program is altered



(a) Running RFM example. (b) Optimization result.

Fig. 1.

compared to a base definition in other SPL transformations, e.g., classes are named differently.

In our running example for this paper in Figure 1a, there is one feature module *F1* which defines the base program of the SPL and which is altered by SPL transformations which follow. That is, there is a number of RFMs, *R2* to *R6* which transform the base program of *F1*. When a user selects feature *F1* and does not select any RFM, the configured program will be a copy of the code of *F1*. When a user selects all features (top-down order), *F1* along with *R2* to *R6* the composed program will expose the functionality of *F1* but will have a different structure. Specifically, when all features are selected, then the resulting code will be a class myQueue with no members and a class Queue with a field _elements and a method pop.

3 Optimizing Refactoring Sequences

We consider two ways to optimize a given sequence of refactorings: optimizing the verification phases and optimizing the transformation phases of the sequenced refactorings. Optimizing *verification phases* in a sequence of refactorings is to check whether preceding refactorings establish preconditions of later refactorings [28,16]. When a refactoring's precondition is satisfied by an earlier refactoring in a sequence, the program does not have to be validated for the latter refactoring, and thus not parsed and traversed *for verification issues* – performance can be gained by fusing verification phases [28,16].

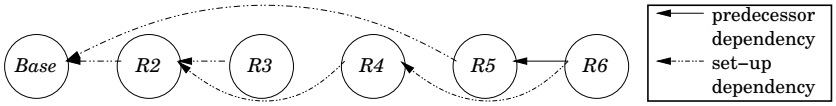


Fig. 2. Optimization steps in running example of Fig. 1a.

Optimizing the *transformation phases* for a sequence of refactorings is to fuse transformations performed by successive refactorings. For example, we can fuse two successive refactorings if both refactorings rename the same class, e.g., $R4_{\mathsf{TestList}\mapsto\mathsf{ArrayList}} \bullet R2_{\mathsf{List}\mapsto\mathsf{TestList}}$ can be replaced by $C2_{\mathsf{List}\mapsto\mathsf{ArrayList}}$. As we do not have to traverse the code twice to parse it, to set up the type system, to look for calls to the method, and to update them, we expect performance benefits. The optimizations we propose work without prior code analysis (*algebraic optimization*) and with prior code analysis and cost functions (*cost-based optimization*).

3.1 Algebraic Optimization

In this work, we concentrate on the concepts for fusing *transformation phases* of refactorings to improve composition performance (fusing verification phases has been analyzed before [28,16]). To optimize a given sequence of refactorings, we reorder sequenced refactorings and fuse them finally. We reorder refactorings to group refactorings of which transformation phases could be fused according to fusing rules we define. We identify fusible refactorings by analyzing their parameters and types.

Basic Concept. To optimize the RFM sequence of Figure 1a, we iterate the sequence of refactorings and calculate fusible refactorings. Two refactorings are fusible when there is a set-up dependency between them, the complete precondition of the later refactoring is satisfied by the preceding refactoring, and when the fused refactoring again is a standard refactoring according to [10] or the empty refactoring. For instance, we calculate, that $R2_{\mathsf{List}\mapsto\mathsf{TestList}}$ could be fused with $R4_{\mathsf{TestList}\mapsto\mathsf{ArrayList}}$ to $C2_{\mathsf{List}\mapsto\mathsf{ArrayList}}$ (see Figure 1b) because the output element TestList of *R2* is the input element of *R4* and the fusing result is the standard refactoring Rename Class; the same for *R4* and *R6*. We then try to reorder *R4* and *R6* to become successors of *R2*.

To prevent errors introduced by our reordering, we propose to compute a dependency graph concept for all refactorings. Especially, we look for two kinds of dependencies: (1) *set-up dependencies* toward preceding RFMs where one preceding refactoring sets up some code elements required by a subsequent refactoring, and (2) *predecessor dependencies* toward preceding refactorings where a preced-

Table 1. Excerpt from [20,21]: Fusing rules to optimize RFM sequences.

Preceding RFM	**Following RFM**	**Fused RFM**
Rename $\text{Class}_{\text{C1}\Rightarrow\text{C2}}$	Rename $\text{Class}_{\text{C2}\Rightarrow\text{C3}}$	Rename $\text{Class}_{\text{C1}\Rightarrow\text{C3}}$
Extract $\text{Interface}_{\text{C1}\Rightarrow\text{I2}}$	Rename $\text{Class}_{\text{I2}\Rightarrow\text{I3}}$	Extract $\text{Interface}_{\text{C1}\Rightarrow\text{I3}}$
Rename $\text{Method}_{\text{M1}\Rightarrow\text{M2}}$	Inline $\text{Method}_{\text{M2}}$	Inline $\text{Method}_{\text{M1}}$
Move $\text{Class}_{\text{C1}\Rightarrow\text{C2}}$	Move $\text{Class}_{\text{C2}\Rightarrow\text{C3}}$	Move $\text{Class}_{\text{C1}\Rightarrow\text{C3}}$
Rename $\text{Class}_{\text{C1}\Rightarrow\text{C2}}$	Collapse $\text{hierarchy}_{(\text{C2},\text{C3})\Rightarrow\text{C3}}$	Collapse $\text{Hierarchy}_{(\text{C1},\text{C3})\Rightarrow\text{C3}}$
Extract $\text{Class}_{\text{C1}\Rightarrow\text{C2}}$	Rename $\text{Class}_{\text{C2}\Rightarrow\text{C3}}$	Extract $\text{Class}_{\text{C1}\Rightarrow\text{C3}}$
Extract $\text{Method}_{\text{M1}\Rightarrow\text{M2}}$	Rename $\text{Method}_{\text{M2}\Rightarrow\text{M3}}$	Extract $\text{Method}_{\text{M1}\Rightarrow\text{M3}}$
Extract $\text{Class}_{\text{C1}\Rightarrow\text{C2}}$	Rename $\text{Class}_{\text{C2}\Rightarrow\text{C3}}$	Extract $\text{Class}_{\text{C1}\Rightarrow\text{C3}}$
Extract $\text{Class}_{\text{C1}\Rightarrow\text{C2}}$	Move $\text{Class}_{\text{C2}\Rightarrow\text{C3}}$	Extract $\text{Class}_{\text{C1}\Rightarrow\text{C3}}$
Extract $\text{Subclass}_{\text{C1}\Rightarrow\text{C2}}$	Rename $\text{Class}_{\text{C2}\Rightarrow\text{C3}}$	Extract $\text{Subclass}_{\text{C1}\Rightarrow\text{C3}}$
Extract $\text{Subclass}_{\text{C1}\Rightarrow\text{C2}}$	Move $\text{Class}_{\text{C2}\Rightarrow\text{C3}}$	Extract $\text{Subclass}_{\text{C1}\Rightarrow\text{C3}}$
Extract $\text{Superclass}_{\text{C1}\Rightarrow\text{C2}}$	Rename $\text{Class}_{\text{C2}\Rightarrow\text{C3}}$	Extract $\text{Superclass}_{\text{C1}\Rightarrow\text{C3}}$
Extract $\text{Superclass}_{\text{C1}\Rightarrow\text{C2}}$	Move $\text{Class}_{\text{C2}\Rightarrow\text{C3}}$	Extract $\text{Superclass}_{\text{C1}\Rightarrow\text{C3}}$
Push-Down $\text{Field}_{\text{F1}\Rightarrow\text{F2}}$	Pull-Up $\text{Field}_{\text{F2}\Rightarrow\text{F1}}$	$\varnothing$
Push-Down $\text{Method}_{\text{F1}\Rightarrow\text{F2}}$	Pull-Up $\text{Method}_{\text{F2}\Rightarrow\text{F1}}$	$\varnothing$

ing refactoring requires another refactoring to establish a required deletion. The conceptual dependency graph for Figure 1a is given in Figure 2.[2]

Using the computed dependency graph we *try* to reorder *R4* and *R6* according to their fusibility. However, we only commute refactorings that do not have predecessor dependencies among each other. Further, we update the parameters of two commuted refactorings when both expose set-up dependencies toward the same predecessor refactoring and share fully qualified names. For instance in Figure 2, we commute *R3* with *R4* because *R4* is fusible with *R2*. As *R3* and *R4* both expose set-up dependencies toward *R2* and parameters share the identifier TestList, we update *R3* to become $C3_{\text{ArrayList.get}\mapsto\text{pop}}$, see Figure 1b. However, we do not reorder *R6* because its predecessor dependency towards *R5* disallows commuting with *R5*. Finally, we fuse successive refactorings according to our fusing rules of Table 1, see [20,21] for a complete list.

Name capture. When a method A is renamed by a Rename Method refactoring, all methods that override A or that are overridden by A are renamed accordingly [10]. *Name capture* is an error in refactoring that occurs when methods override each other after a refactoring executed which did not override each other before the refactoring executed [26,25,31]. When commuting refactorings, we may not know whether methods referenced in 2 RFMs will override each other after both RFMs get commuted. Nevertheless, we must guarantee that we do not introduce name capture, i.e., that the optimized refactoring sequence still produces *the same* program. Name capture must also be prevented for fields.[3]

We present three concepts which avoid name capture. In concept #1, we track which refactoring parameter (fully qualified name) emerges out of which

[2] *R2* creates class TestList which is required by *R3* and *R4*. *R4* creates class ArrayList which *R6* requires to exist. *R5* removes Queue which *R6* requires to not exist.

[3] A special situation which disallows commuting refactorings occurs when a Move Method RFM or Inline Method RFM follows an Extract Interface RFM and both operate the same class.

code element in the base code. By analyzing relationships between the code elements in the base code we can then decide whether two elements override each other. In concept #2, we disallow commuting of two refactorings when both reference methods, e.g., Rename Method refactorings, or when both reference fields. However, we only must disallow commuting when field or method names match in the refactorings to be commuted. In concept #3, we define all the elements, which a refactoring alters, inside feature modules. As a result, we know all (overridden) methods which are effected by a Rename Method RFM.

Heuristical reordering. Reordering *itself* can produce performance benefits for the composition process. For example, when a Rename Field RFM follows an Encapsulate Field RFM[4], then reordering is beneficial though both RFMs cannot be fused. The reason is that the field to be renamed might be referenced multiple times in the transformed code but is only referenced once in the getter and once in the setter method after encapsulating the field. When the field name is integrated in the generated method names, this optimization is not possible. For caution in this case, we should disallow commuting Encapsulate Field RFMs with any other refactoring transforming the field.

Commuting a Hide Method refactoring[5] with a Rename Method refactoring is beneficial. After hiding the method, the composer can reason on the new visibility qualifier of this hidden method and thus can prune the code traversed for renaming. For example, if hiding the method push produces a private method then for renaming push the composer just must traverse the class as no references outside this class can exist. Similar optimizations are possible for fields.

Random reordering. With the concepts presented so far, we cannnot optimize the sequence $R3_{RenameClass:\mathsf{C3}\mapsto\mathsf{C4}} \bullet R2_{MoveClass:\mathsf{C2}\mapsto\mathsf{C3}} \bullet R1_{RenameClass:\mathsf{C1}\mapsto\mathsf{C2}}$ because we cannot detect any fusibility. We can fuse neither *R1* with *R2* nor *R2* with *R3* because the resulting refactoring would be no standard refactoring – fusing them would exceed our set of operations.[6] We also do not detect fusibility between *R1* and *R3* because the output identifier of *R1* does not match the input identifier of *R3*. If we at random commute *R1* with *R2* or *R2* with *R3* then a *new* fusing chance emerges between (reordered) *R1* and (reordered) *R3*. We envision to generate sets of refactoring plans during algebraic optimization also by *randomly* reordering refactorings and to select the shortest plan for execution.

[4] Encapsulate field adds get and set methods for the field to encapsulate [10]. After that, the refactoring transformation replaces every reference to the field by a call to either the get or set method.

[5] Hide Method refactoring reduces the visibility of the method as far as possible [10].

[6] We could provide composite refactorings which do renaming and moving within one step (as shown before [16]) but we refrained due to the infinit number of possible refactoring combinations [16].

3.2 Cost-based Optimization

We can analyze the code to be refactored to estimate the execution costs for individual refactorings. From there we can further optimize a refactoring sequence or select between alternative sequences. We envision to identify refactorings which alter distinct parts of a program (possibly, distinct artifact types). If we can reorder these refactorings to succeed each other, we can *parallelize* their execution, i.e., we can load and alter the distinct program parts in parallel. One approach to identify relevant distinct program parts is to collect visibility qualifiers and inheritance hierarchies from the program to refactor. If then the visibility of two code elements is very restricted, e.g., private or protected, and both occur in different classes (hierarchies) according refactorings perform on distinct pieces of code and can be executed in parallel.

As an example, consider the Rename Method refactorings $R3_{\mathsf{TestList.get}\mapsto\mathsf{pop}}$ and $R7_{\mathsf{myQueue.pop}\mapsto\mathsf{insert}}$ where both methods are analyzed to not override each other or a common method, and to be qualified as protected. Thus, *R3* and *R7* transform distinct parts of a program. In that case we can infer a parallelization chance and try to make both refactorings successors. We then can load TestList and myQueue and their subclasses in parallel and execute *R3* and *R7* in parallel as shown in Figure 3. We can also parallelize refactorings which transform private members of different classes.

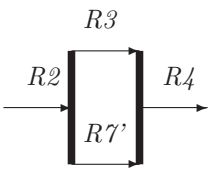


Fig. 3. Parallel RFM execution.

If the visibility of members is private or protected and – in the latter case – the inheritance hierarchy is small, then according refactorings are expected to be cheap. By deferring presumably expensive refactorings we increase the chance that for a longer time the memory is not exceeded by loading code. This reduces the number of buffer misses (increases performance).

4 Case Studies

We implemented the basic concept of algebraic optimization of RFMs prototypically. Currently, a *separate* optimizer operates RFMs in a step separately *before* the composer tool runs. Details can be found in [20,21].

4.1 Study Setup

We took programs of different size and purpose as study objects. We composed the feature modules and took the composer's runtime. Then we run our optimizer tool and took its runtime, too. Finally, we composed the optimized sequences and compared the composer's runtime to the time of the unoptimized composition,

Table 2. Measured tool run times (in *ms*).

Program	#SLOC*	#RFMs (unopt.)	#RFMs (opt.)	Composer (unopt.)	Composer (opt.)	Optimizer	Pure Optimization
Simple List (a)	19	5	2	12018.6	9870.4	8934.6	9.4
Simple List (b)	19	8	4	12840.7	9546.8	9401.4	9.5
Simple List (c)	19	10	4	16359.3	10412.3	9074.6	20.3
TankWar	∼1K	10	4	31934.2	14093.6	8206.3	18.8
Workbench.texteditor (a)	∼16K	10	4	172162.4	83561.1	18749.9	17.2
Workbench.texteditor (b)	∼16K	17	3	253831.2	59731.2	18448.4	23.3
Workbench.texteditor (c)	∼16K	55	3	769617.5	61292.1	77632.7	101.4
ZipMe	∼3K	3	3	20461	20281.4	7867.1	10.8

*lines of source code without RFMs

see Table 2.[7] As we do not change the composition of non-RFM features we prune the studies to only have one non-RFM feature module *F1* each.

Simple List. As a proof of concept we applied three different sequences of RFMs to a *conceptual* list implementation. In these sequences, we fuse an RFM, which extracts the interface AbstractList from class List, with reordered RFMs which all rename the extracted interface. As a result one new refactoring is generated which extracts the interface with the final name in the first place. In another sequence, we could not fuse refactorings which had fusing chances due to a predecessor dependency.

TankWar. We analyzed TankWar an SPL of arcade games for desktop computer and cell phone developed prior to this evaluation at Magdeburg University. The study is still small-scale but provides functionality (in contrast to the Simple List case).

Workbench.texteditor. In order to analyze the performance effect of optimizing RFM sequences, we should pay attention to the size of the transformed program. For that, we reused a large-scale study of the Eclipse[8] library *workbench.texteditor* from prior work [19]. To this library, we applied three different sequences of RFMs with lengths ranging from 10 to 55 RFMs.

[7] We used a Microsoft Windows XP Home Edition SP2 on an Intel® Core™2 CPU T5500 @ 1.66GHz, 667MHz FSB, 0.99 GB RAM. The given measurements are *averages* of 10 runs, listed one-by-one in [21]. The optimizer solely generates optimized and unchanged RFMs into a folder *Optimized* but does not copy other SPL feature modules. To measure the composition performance for the optimized RFM sequence, we manually copied the other feature modules into the *Optimized* folder. Detecting name capture is not yet implemented. We implemented five fusing rules.

[8] http://www.eclipse.org/

ZipMe. We finally analyzed a study of a compression library ZipMe from prior work [18] which showed us that our optimization effort may be worthless and, thus, derogatory. That is, in the ZipMe study, there is no fusing chance and thus, the runtime of our optimizer tool directly increases composition time.

In Table 2, we summarize the measured runtimes of the optimizer tool as well as the runtimes of the composer tool on the unoptimized and on the optimized RFM sequences. In some cases we gained performance increases, e.g., for case Workbench.texteditor (c) we gained a performance benefit of 81% through optimization. In many cases, however, the overall composition time increased with optimization, i.e., performance decreased. For example, the unoptimized composition time for case Simple List (a) is 12018.6ms and the optimized composition time plus the optimizer runtime is 18805ms, i.e., a performance loss of 56%. Nevertheless, we did not fail optimizing. The increased composition time is mainly caused by the optimizer prototype operating independently from the composer tool. Times for loading RFMs, thus, contribute to both the composer *and* the optimizer tool, and times for writing optimization results arise. When the optimizer is integrated with the composer tool (possible future work), RFMs would be loaded only *once* and the need to write the optimized RFM sequence to harddisk vanishes. To respect this, we also measured the time for purely optimizing loaded refactoring sequences. When considering pure optimization time, we get a significant performance benefit for all cases but the ZipMe case.

From the measurements we observed that the performance benefit increases with a growing size of the program to be transformed, the highest performance benefits were measured for the biggest program (Workbench.texteditor). We also observed that with a growing number of fusible RFMs, the optimization benefit increases, too. In the case of ZipMe, the optimizer could not produce a benefit and, thus, for this case optimization effort is derogatory.

Threats to Validity. The measurements and benefits depend on the loading of RFMs. If to load an RFM takes a long time, reducing the number of loads saves a lot time. The measurements and benefits further depend on the execution time for a single RFM. If executing a single RFM takes a long time, reducing the number of executions saves a lot time. The RFM composer tool we used (the only one we know of) is written for flexibility and not for performance. Thus, for other RFM composers the numbers may be different.

Our approach may remove whole subsequences from a sequence of transformations. This implicitly removes precondition checks of removed transformations. We did not distinguish composer run times for checks and actions because we did not change the composer.

5 Related Work

Researchers composed transformations, refactorings, and their preconditions before, e.g.,[28,16,8,15,13], and formalized refactorings and their preconditions [25].

In contrast to prior work, we *reorder, replace, and update* (sequences of) refactorings in a transformation sequence to create a faster-to-execute sequence of *standard* refactorings. To the best of our knowledge, this is new.

Dig fuses sequences of refactorings [9, p.95], sequences which were recorded independently on the same program. He adapts parameters of the refactorings in order to sequentialize the according refactorings. Similarly, Lynagh fuses concurrent edits on code and resolves conflicts by commuting and reverting edits [22]. It may happen that by fusing sequences of edits and refactorings, the resulting sequence may execute faster than their concatenation (in case this concatenation works at all). In contrast to prior work, we *intend* to shrink a *single* sequence and for that fuse refactorings and reorder them.

Design maintenance systems (DMSs) organize transformations a program was built from [5]. DMSs commute transformations, update transformation parameters [5, p.175], and replace subsequences of transformations [5, p.179] in order to integrate new transformations and to evolve the program [5, p.179ff]. DMSs compose transformations [5, p.129] but no specific rules are given for how to do so. We reorder refactorings to fuse them for performance reasons and to remove superfluous transformations (not in the focus of Baxter [5, p.276]).

Researchers describe how to calculate dependencies between transformations in general and refactorings in particular [23,24]. We also compute these dependencies, so prior research is a basis for our research. Based on dependencies between refactorings, we introduce fusing rules for transformation phases of refactorings. Further, we discussed optimizations of refactoring sequences based on code analyses (cost-based optimization, cf. Sec. 3.2).

Pérez derives refactoring sequences that minimize code smells [27]. We transform given sequences of refactorings with fusing rules in order to yield performance benefits for their execution. We cannot imagine how to *safely* automate the restructuring process of a program towards a given interface.

In our approach, we know generics of all SPL transformations and thus the transformation's effects, i.e., we know the effects of refactorings as defined in RFMs. In general, however, developers may be allowed to define transformations/rewrites beyond refactorings, e.g., [6]; transformations of which we maybe do not know the effects. In those cases, our fusing rules do not apply, as we maybe do not know the effects of transformations before executing them.

Relational algebra organizes a set of algebraic operations users can execute on databases [7,11]. With SQL, a user describes *declaratively* the data she needs [30]. The algebra expression translated from the declarative query may be suboptimal and thus it is optimized *algebraically* (without table analyses) and *cost-based* (with table analyses) [12,11,30]. In distributed database management systems, a query result can be computed on different systems in parallel to improve query time [1,7,12]. When generating our SPL program, the selected features are translated into sequenced program transformations – a sequence which may be suboptimal. In this paper, we showed how a sequence of refactorings inside RFMs can be optimized algebraically and cost-based, i.e., without and with analyzing the code to refactor. In our envisioned cost-based optimization we parallelize

RFMs to improve composition time which then will closely correlate to parallel database management systems. However, database management systems do not organize program transformations.

Batory et al. related program transformations to category theory and, thus, sketched the formal basis of our optimizations [2,3]. Our fusion rules and heuristical reordering for refactoring transformations thus implement these abstract concepts. In addition, we presented ideas on cost-based optimizations of refactoring sequences.

6 Conclusions

Product line users tailor programs by selecting features. Selected features can translate into program transformations which execute sequentially on a base program. A sequence translated directly from a user selection can be inefficient. In this paper, we showed how to optimize sequences of refactoring transformations to reduce the composition time of product line programs. We presented a prototype and evaluated it in several case studies. We observed that the optimization reduces the time to compose a program in most cases though not all.

References

1. P.M.G. Apers, A.R. Hevner, and S.B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering*, 9(1):57–68, 1983.
2. D. Batory. A modeling language for program design and synthesis. *Lecture Notes in Computer Science*, 5316:39–58, 2008.
3. D. Batory. Using modern mathematics as an fosd modeling language. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 35–44, 2008.
4. D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
5. I.D. Baxter. *Transformational maintenance by reuse of design histories.* PhD thesis, 1990.
6. M. Bravenboer, K.T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.
7. S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Symposium on Principles of Database Systems*, pages 34–43, 1998.
8. M. Ó Cinnéide and P. Nixon. Composite refactorings for java programs. In *Workshop on Formal Techniques for Java Programs*, pages 129–135, 2000.
9. D. Dig. *Automated upgrading of component-based applications.* PhD thesis, University of Illinois at Urbana-Champaign, 2007.
10. M. Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., 1999.
11. P.A.V. Hall. Optimization of single expressions in a relational data base system. *IBM Journal of Research and Development*, 20(3):244–257, 1976.
12. M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

13. P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. *Electronic Notes in Theoretical Computer Science*, 57:144–162, 2001.
14. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
15. G. Kniesel. A logic foundation for program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, 2006.
16. G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, 2004.
17. C. W. Krueger. New methods in software product line practice. *Communications of the ACM*, 49(12):37–40, 2006.
18. M. Kuhlemann, D. Batory, and S. Apel. Refactoring feature modules. In *Proceedings of the International Conference on Software Reuse*, pages 106–115, 2009.
19. M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 177–186, 2009.
20. M. Kuhlemann, L. Liang, and G. Saake. Algebraic and cost-based optimization of refactoring sequences. Technical Report 5, Faculty of Computer Science, University of Magdeburg, 2010.
21. L. Liang. Optimizing sequences of refactorings. Master thesis, University of Magdeburg, Germany, MAR 2010. http://wwwiti.cs.uni−magdeburg.de/iti_db/publikationen/ps/auto/thesis-Liang.pdf.
22. I. Lynagh. An algebra of patches. http://urchin.earth.li/∼ian/conflictors/paper-2006-10-30.pdf, 2006.
23. T. Mens, G. Kniesel, and O. Runge. Transformation dependency analysis - a comparison of two approaches. In *Actes des journéees Langages et Modèles à Objets*, pages 167–184, 2006.
24. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.
25. T. Mens, N. v. Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
26. W.F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
27. J. Pérez. Enabling refactoring with HTN planning to improve the design smells correction activity. In *BElgian-NEtherlands software eVOLution workshop*, 2008.
28. D.B. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
29. N. Siegmund, M. Kuhlemann, S. Apel, and M. Pukall. Optimizing non-functional properties of software product lines by means of refactorings. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*, pages 115–122, 2010.
30. J.M. Smith and P.Y.-T. Chang. Optimizing the performance of a relational algebra database interface. *Communications of the ACM*, 18(10):568–579, 1975.
31. P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. *ACM SIGPLAN Notices*, 31(10):268–285, 1996.
32. C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 130–139, 2003.