Reduction of Program-generation Times by Transformation-sequence Optimization

Martin Kuhlemann, Andreas Lübcke and Gunter Saake
University of Magdeburg, Magdeburg, Germany

Keywords: Program Generation, Code-transformation Time, Optimization.

Abstract: Transforming source code is common today. Such transformation process may involve the execution of a number of program transformations on the source code. Overall, the transformation process can last long when individual program transformations last long and when high numbers of program transformations need to be executed. In this paper, we introduce and discuss techniques that can reduce the time transformation tools need to produce a program.

1 INTRODUCTION

A number of mainstream programming languages involve the translation of source code. Such translation is possible by consecutively executing a number of program transformations (Batory et al., 2004; Schaefer et al., 2010; Kuhlemann et al., 2009; Baxter, 1990). Generally, such transformation process might last long because (a) individual transformations might last long (e.g., the linking of the Mozilla Firefox browser can last more than 30 minutes), (b) the number of program transformations might be high, or (c) a combination of both reasons. Such transformation process even might exceed memory.[1] Thus, we investigate the optimization of transformation sequences such that similar problems do no longer occur. We argue that such problems are especially harmful in the following sample scenarios:

- In configurable programs (sometimes called product lines (Czarnecki and Eisenecker, 2000)), program transformations can be executed in sequences in order to generate programs with certain features (Batory et al., 2004). If programs have many features, many program transformations must be executed.
- In step-wise refinement (Wirth, 1971), program transformations implement/encapsulate (unforeseen) evolutionary steps and might have not been analyzed for being necessary with respect to the finally generated program. If many evolutionary steps exist, so there are many program transformations to execute.

While the problem of long execution times for program-transformation sequences has been observed before (Baxter, 1990; Batory, 2007; Kuhlemann et al., 2010), no precise rules were proposed for general program transformations in order to reduce the times that tools need for executing sequences of these program transformations. In this paper, we discuss new ideas of how to reduce the runtimes of tools, which execute sequences of program transformations. Future work remains to evaluate and detail these first ideas in industrial environments and tools.

2 BACKGROUND ON TRANSFORMATION SYSTEMS

Different types of program transformations are common, *superimposition* and *pattern-based* transformations.

2.1 Superimposition Transformations

Jak, FeatureC++, and FeatureHouse are languages that extend programming languages by superimpositions (e.g., Jak extends Java by superimpositions) (Batory et al., 2004; Apel et al., 2005; Apel et al., 2009).[2] A superimposition is a program trans-

[1] https://developer.mozilla.org/en/Building_with_Profile-Guided_Optimization (accessed: December 23, 2011).

[2] FeatureC++ also supports additional types of transformations.

DOI: 10.5220/0004093901820186
In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2012), pages 182-186
ISBN: 978-989-8565-13-6

Program transformation *BaseElem*

```
1  public class Element {
2    public String name;
3    public void print(){
4      ...
5      System.out.print(name);
6    }
7  }
```

Program transformation *NeighborElem*

```
8  refines class Element {
9    private String neighbor;
10   private void printNeighbor(){...}
11   public void print() {
12     printNeighbor();
13     Super.print();
14   }
15 }
```

(a) Jak transformations.

```
16 public class Element {
17   public String name;
18   public void print(){
19     printNeighbor();
20     ...
21     System.out.print(name);
22   }
23   private String neighbor;
24   private void printNeighbor(){...}
25 }
```

(b) Result of executing the transformations of Fig. 1a.

Figure 1: Jak program transformations and their execution result.

formation, which is structured like its input program; code of such superimpositions is added to the input program in positions that coincide with the code position inside the superimposition.

In Figure 1, we list two superimposition transformations written in Jak, *BaseElem* and *NeighborElem*. *BaseElem* takes a program as an input and adds a class Element to it, which has a field name and a method print. *NeighborElem* takes a program as an input (possibly the program generated by *BaseElem*) and adds the members neighbor and printNeighbor to the class Element (keyword refines, Line 8); further, the *NeighborElem* method print overrides the method print of Element (as the methods' positions coincide) – thereby, print of *NeighborElem* calls the *BaseElem* method print (keyword Super, Line 13). When we execute the program transformations of Figure 1a, we generate the program of Figure 1b.

```
1  public class Element {
2    public String name;
3    public void print(){
4      ...
5      System.out.print(name);
6    }
7  }
```

```
8  public aspect ElementAspect {
9    private String Element.neighbor;
10   private void Element.printNeighbor(){...}
11   before() : execution(void Element.print()) {
12     printNeighbor();
13   }
14 }
```

Figure 2: AspectJ aspects and a program they alter.

2.2 Pattern-based Transformations

AspectJ and macro languages (like the one of C++) extend programming languages by program-transformation-like mechanisms (Kiczales et al., 2001; Stroustrup, 1991). Basically, the mechanisms in these languages take a program as an input, detect code that follows an explicitly defined pattern (be the pattern (a) a description of code positions or (b) a piece of code of the extended program), and add/replace code according to the matching code. For example, aspects might match the method of a certain class and transform it.

In Figure 2, we show a base program consisting of class Element and we show an aspect ElementAspect. The aspect adds members to class Element of the base program (neighbor and printNeighbor to class Element) and extends method print of Element. The result of executing the aspect ElementAspect with its base program is equal to the program of Figure 1b. The pattern, which matches the method print (Fig. 2, Line 11), is specific to the base program as it depends on that there is a class Element and on that there is a method print inside the Element class. However, the pattern might also contain wild cards to match different methods; for example, the pattern execution(void Element.*()) can match multiple methods of the Element class in order to extend them.

3 OPTIMIZING SEQUENCES OF PROGRAM TRANSFORMATIONS

We have introduced basic techniques of transformation systems in Section 2. Now, we analyze new

Table 1: Support of optimization concepts by transformation types.

Transformation type	**Postponing**	**Locality**	**Overwritten**	**Parallelism**
Superimposition transformations	⊕	⊙	⊕	⊕
Pattern-based transformations	⊖	⊙	⊕	⊙

⊕good support; ⊙problematic support; ⊖bad/no support

approaches to reduce the time tools need to execute sequences of program transformations; specifically, these approaches involve the *postponing of long-lasting transformations*, the analysis of *locality of altered code*, the *removing of overwritten program transformations*, and the *parallel processing of program transformations*. In Table 1, we summarize the support for the concepts we propose by the transformation types.

Postponing of Long-lasting Transformations. We propose to postpone in sequences those transformations which last long; we do so to execute short-term transformations first. First, this helps to not run out of memory when less objects must be kept in memory, which in turn helps to reduce the execution time of a program-transformation tool. Second, postponing long-lasting program transformations can help to abort a generation process early when a transformation inside a sequence to execute is in error (as more transformations are executed earlier). For example, if transformation *A* takes long but not transformation *B*, then *B* should be executed before *A*; if *B* fails then it does before executing the long-lasting *A* and when *A* fails it does after the short-term *B*.

In order to reorder two program transformations *A* and *B*, both transformations must be commutative (i.e., $A(B(program)) = B(A(program))$). However, two transformations might not be commutative when there are interdependencies between them (Mens et al., 2006; Mens et al., 2007; Whitfield and Soffa, 1997). For example, in Figure 1a, we must attend interdependencies between the program transformations *BaseElem* and *NeighborElem* such that *BaseElem* must execute before *NeighborElem*; specifically, *NeighborElem* extends a class Element, which thus must exist, and the method extension of *NeighborElem*, print, calls the base method, which only exists if *BaseElem* executed before *NeighborElem*. We further cannot postpone and reorder transformations if we cannot always detect which pieces of code they affect (i.e., when program transformations do not enumerate these pieces).

Now, how can we estimate the time, which a tool needs to execute a program transformation? We propose to analyze program transformations with respect to the number of pieces of code, which each transformation affects. We assume that transformations, which alter more pieces of code, last longer and should thus be postponed in sequences as much as possible. We also could apply additional code metrics (e.g., metrics measuring positional distances of code elements inside respective input programs) to estimate the time of executing a transformation. For superimposition languages, we can count the number of pieces of code that are altered by a program transformation. For example, the feature transformation *NeighborElem* in Figure 1a alters four pieces of code (Element, Element.neighbor, Element.printNeighbor(), Element.print()) whereas *BaseElem* in this figure alters only three pieces of code (Element, Element.name, and Element.print()). We thus can assume that executing *BaseElem* before *NeighborElem* (if possible with respect to the transformations' interdependencies) keeps the memory non-full for a longer period of time which in turn improves performance of the program-generation process. Program transformations, which do not enumerate the pieces of code they alter, can hardly be estimated with respect to the number of pieces of code they alter. Summarizing, we can estimate well the time to execute superimposition transformations (as they list all the pieces of code they alter) but not pattern-based transformations.

Locality of Altered Code. We could group transformations, which alter pieces of code that are near each other with respect to the code structure because this could allow us to pass intermediate transformation results between the transformation steps in memory (if they are not grouped, buffer management might force tools to write intermediate results to hard disk). This way, we might reduce disk accesses and thus improve performance. For example, we could group transformations which target methods of the same class such that this class and classes, which use this class or are related to this class, only must be loaded, parsed, and processed once while other classes (transformed by other transformations) need not be loaded for the time of executing these transformations. For superimposition transformations and pattern-based transformations, locality of code might be hard to detect from the transformation description (e.g., when transformed pieces of code are scattered across classes of which relations are unknown).

Removing Overwritten Program Transformations. In many program-transformation languages, transformation effects can be undone or overwritten by program transformations that are executed later. For example, DELTAJAVA (Schaefer et al., 2010) is a program-transformation language with superimposition-like mechanisms, which allow program transformations to add code, to alter code, and to remove code. If we can detect program transformations that overwrite effects of other transformations, we might be able to omit these other transformations without affecting the transformation result. To implement this approach, we can analyze the containment of pieces of transformed code. For example, if we can detect that a first program transformation solely creates a method inside a class and a later program transformation removes or replaces this class (including the method) we can omit the first transformation without problems. In accordance to database technology, we call such cases *blind-write cases*. We might also alter transformations when transformed pieces of code of two transformations just partly overwrite each other.

In Jak, pieces of code with equal scoped names overwrite each other. If a piece of code overwrites another piece of code but does neither call nor reference this other piece of code (using refines or Super); this other piece gets replaced and needs not to be inserted in the first place.

We can detect containment and overlapping of transformed pieces of code of two transformations straightforwardly for superimpositions because superimpositions enumerate respective pieces. For pattern-based program transformations, we must compare the patterns and maybe can neither detect containment nor overlapping of transformed pieces of code at all.

Throughout this optimization, we must check that program transformations do not depend on a piece of code, which was generated by a transformation to be removed, as they could fail after the removal.

Parallel Processing of Program Transformations. With superimposition transformations, we easily can determine program transformations that alter disjoint pieces of code in a program. This is because superimpositions enumerate the pieces of code they transform and we can compare these pieces. Once, we know that transformations alter disjoint pieces of code in a program, we can execute both transformations in parallel (after making them successors in the transformation sequence) and merge their results. Executing pattern-based transformations in parallel is hard but possible (as shown for refactorings before (Kuhlemann et al., 2010)). That is, though we can execute such transformations in parallel, we must ensure that the merge of the result is equal to the result of executing the transformations sequentially. In accordance to database technology, we call problems in such cases *lost-update errors*.

4 RELATED WORK

Batory et al. motivated the need to optimize the program-generation process for superimposition transformations (Batory, 2007; Batory, 2004). We now gave rules that shall allow tools to estimate the time they need for executing a program transformation. In previous work, we gave such precise rules only for sequences of refactoring transformations (Kuhlemann et al., 2010). Others optimized sequences of program transformations (mostly refactorings) by targeting the precondition checks of these program transformations (Cinnéide and Nixon, 2000; Kniesel and Koch, 2004; Kniesel, 2006; Roberts, 1999); we did not aim at reducing the precondition checks of program transformations to increase transformation-tool performance.

Approaches exist, to reorder program transformations of concurrent program-transformation sequences (Lynagh, 2006). In our approach, we reorder program transformations in individual program-transformation sequences.

Other approaches involve the reordering of transformations in individual program-transformation sequences (Dig, 2007; Dig et al., 2008; Baxter, 1990); these approaches, however, do not have the purpose of reducing any program-generation time. Instead, they reorder program transformations, for example, in order to synchronize maintenance edits made to a program with a previous version of this program in a versioning system.

5 CONCLUSIONS

In this paper, we analyzed how sequences of program transformations can be executed faster. To this end, we introduced techniques to detect program transformations, which perform unnecessary operations. Further, we introduced ideas to estimate the time tools need to execute a program transformation. As a result, our techniques will help to reduce the time tools need to execute sequences of program transformations as well as resources like memory.

There is a lot of future work for us to do. First, we need to perform studies in order to verify our concepts (e.g., regarding the effort estimation for program transformations). That is, for example, we need to check whether the count of pieces of code to transform really is a good indicator for the time tools need to execute transformations. Second, we need to investigate on the effect we can gain in real-world programs. Third, we need to investigate new cost models for program transformations (just as they exist in database systems for data transformations). Finally, we need to find out how program transformations should look like in order to apply our concepts to them. We plan to adopt techniques of query optimization and transaction management of database management systems for program-transformation tools in this future work.

REFERENCES

Apel, S., Kästner, C., and Lengauer, C. (2009). FeatureHouse: Language-independent, automated software composition. In *Proceedings of the International Conference on Software Engineering*, pages 221–231. IEEE Computer Society.

Apel, S., Rosenmüller, M., Leich, T., and Saake, G. (2005). FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the Conference on Generative Programming and Component Engineering*, pages 125–140. Springer Verlag.

Batory, D. (2004). The road to Utopia: A future for generative programming. In *Proceedings of the Seminar on Domain-Specific Program Generation*, pages 211–250. Springer Verlag.

Batory, D. (2007). A modeling language for program design and synthesis. In *Proceedings of the Lipari Summer School on Advances in Software Engineering*, pages 39–58. Springer Verlag.

Batory, D., Sarvela, J., and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371.

Baxter, I. (1990). *Transformational maintenance by reuse of design histories*. PhD thesis, University of California at Irvine, USA.

Cinnéide, M. Ó. and Nixon, P. (2000). Composite refactorings for Java programs. In *Proceedings of the Workshop on Formal Techniques for Java Programs*, pages 129–135.

Czarnecki, K. and Eisenecker, U. (2000). *Generative programming: Methods, tools, and applications*. Addison-Wesley Longman Publishing Co., Inc.

Dig, D. (2007). *Automated upgrading of component-based applications*. PhD thesis, University of Illinois at Urbana-Champaign, USA.

Dig, D., Manzoor, K., Johnson, R., and Nguyen, T. (2008). Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 327–353. Springer Verlag.

Kniesel, G. (2006). A logic foundation for program transformations. Technical Report IAI-TR-2006-1, University of Bonn, Germany.

Kniesel, G. and Koch, H. (2004). Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51.

Kuhlemann, M., Batory, D., and Apel, S. (2009). Refactoring feature modules. In *Proceedings of the International Conference on Software Reuse*, pages 106–115. Springer Verlag.

Kuhlemann, M., Liang, L., and Saake, G. (2010). Algebraic and cost-based optimization of refactoring sequences. In *Proceedings of the Workshop on Model-Driven Product Line Engineering*, pages 37–48. CEUR-WS.org.

Lynagh, I. (2006). An algebra of patches. [Available online: http://urchin.earth.li/∼ian/conflictors/paper-2006-10-30.pdf; accessed: July 16,2011].

Mens, T., Kniesel, G., and Runge, O. (2006). Transformation dependency analysis - A comparison of two approaches. In *Proceedings of Langages et Modèles à Objets*, pages 167–184. Hermes Science Publishing.

Mens, T., Taentzer, G., and Runge, O. (2007). Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285.

Roberts, D. (1999). *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, USA.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). Delta-oriented programming of software product lines. In *Proceedings of the Software Product Line Conference*, pages 77–91. Springer Verlag.

Stroustrup, B. (1991). *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition.

Whitfield, D. and Soffa, M. (1997). An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227.