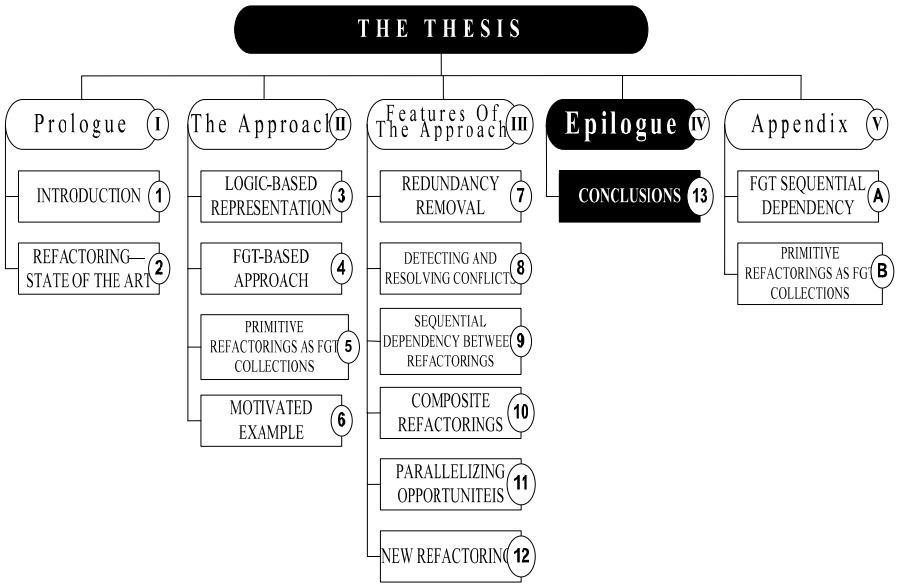
Part IV

Epilogue

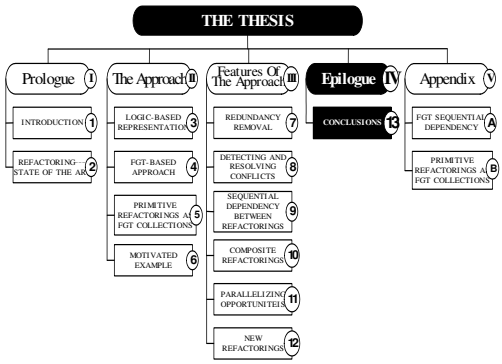


Chapter 13

CONCLUSIONS

13.1 Summary

This work can be summarized as the follows:



A. In part I—which includes chapter one and two—an introduction to refactoring, problems associated with it, and proposed solutions are discussed. A survey of previous work in refactoring topics related to the thesis was presented. The role of evolution in the system life cycle, the levels of the system artifacts where the refactorings can be applied, and the different refactorings formalism techniques were covered in the survey.

B. In part II—which includes chapters three to six—a new formalism to represent refactorings at the design level is presented. The new formalism defines and executes model refactorings as a set of FGTs ordered in one or more FGT-DAGs. It also introduces refactoring pre- and postcondition conjuncts at two different levels (FGT-level and refactoring-level). Detailed descriptions of the set of FGTs that are used in the approach together with their set of preconditions are also presented. A logic-based representation was presented in this part of UML class diagrams, of related objects, and of reference information extracted from the code-level of the system under consideration. The part also discussed the relationship between the proposed FGT paradigm and primitive- as well as composite refactorings. It was shown that FGTs can be the core of a refactoring system in which a wide range of refactorings can be constructed and represented by a collection of these FGTs. To show the feasibility of the approach and its ability to represent refactorings, FGT representations of twenty-nine common primitive refactorings were presented in chapter 5. The chapter also discussed the set of precondition conjuncts of each refactoring and how these precondition conjuncts are related to the precondition conjuncts of their associated FGTs. At the end of this part, in chapter 6, a motivated example was given. The example, "*A simulation of a Local Area Network (LAN)*", is frequently used for teaching refactoring. The chapter shows how the UML class diagram of the LAN system (with the

additional reference information) is represented as logic-terms. In addition to the twenty-nine primitive refactorings presented in chapter 5, chapter 6 shows how two other well-known composite refactorings (**encapsulateAttribute** and **createClass**) are represented in the proposed formalism.

C. **In part III**—which includes chapters seven to twelve—various features of the proposed formalism were explored. Chapter 7 showed how redundancy between FGTs in the same FGT-DAG can be removed. For that a **reduction** algorithm was developed. This feature reduces the number of FGTs and the associated number of refactoring precondition conjuncts, thus increasing the efficiency of refactoring. In addition, the number of sequential dependencies between the different FGTs inside the refactoring will be reduced and the pseudo-conflicts will be eliminated. Chapter 8 showed how conflict freedom can be established using the **detectResolveConflict** algorithm that was developed. Three different kinds of conflicts between pairs of refactorings were described and treated: ordering-conflicts (where conflict can be resolved by ordering one of the refactorings before the other); cancelling-conflicts (where conflict can only be resolved by withdrawing one of the refactorings); and removable-conflicts (where conflicts can be resolved by appropriately modifying FGTs that comprise one of the refactorings). Then, in chapter 9, finding the sequential dependency between two refactorings was discussed. For that a **sequentialDependency** algorithm was developed. Also in this chapter, the deadlock and the ambiguity terms were introduced and treated properly. An FGT-based approach to deal with composite refactorings was introduced in chapter 10. The scope for parallelizing FGT-based refactoring at various levels was discussed in chapter 11. Parallelizing suggestions (extensions) for the different algorithms presented in the thesis were explored in overview. Finally, in chapter 12, the feature of giving the end users the ability to create their own FGT-based refactorings without having to write a code is presented. The proposal of developing a DSL that is based on the FGT paradigm was made, and an illustration was provided of how FGTs can be used to build a refactoring that does not consist of a number of primitive refactorings.

Over the past few years, various aspects of this work have been published in peer-reviewed conferences and workshops:

1. [73] represents the initial work that led to the investigation of the new refactoring formalism using the FGT paradigm. The FGT methodology was briefly introduced in the paper.

7. [74] presents the algorithm to find automatically the optimal ordering in which to apply a batch of refactorings. The proposed algorithm detects implicit sequential dependencies, resolves conflicts between the different refactorings in the batch and minimizes the number of refactoring operations by removing the redundant ones. The algorithm is based on the FGT paradigm described thoroughly in this work.

2. [75] extends the work done in [73] by introducing a new formal definition of refactorings that supposed to work at the UML class diagrams. Feasibility and features of the new approach are explored in the paper.

In the next section, we provide a conclusion to our work.

13.2 Conclusions

The followings has been achieved in this thesis:

1. **FGT-Based Refactoring Formalism Technique:** This work has established a new technique to formalize refactorings applied at the design level (*UML class diagrams in specific*). The new formalism is based on the so-called FGT paradigm. The feasibility and features of the new approach are discussed thoroughly in the thesis. A detailed set of FGTs together with their set of precondition conjuncts were defined in the work. These FGTs are at the core of the refactoring formalism. Based on the new formalism, a design level refactoring can be seen as:

   > *"A collection of FGTs ordered in one or more FGT-DAGs with a set of pre- and postcondition conjuncts installed at the level of the whole refactoring and a set of pre- and postcondition conjuncts installed at the level of each FGT"*

   Several common refactorings already available in the literature (twenty-nine primitive refactorings and two composite refactorings) have been presented in terms of such FGT-DAGs.

2. **Logic-Based UML Class Diagrams Representation:** The work has shown how UML class models can be represented as a set of logic-terms (*facts in Prolog*). The proposed representation can be used for refactoring (as done in this thesis). However, it can also be used as a basis for issuing Prolog queries about a UML system.

3. **Remove Redundancy:** The work has defined a method for removing redundancy at the FGT-level in refactorings that may grow complex as they composed into ever-more larger ones over time.

4. **Detect and Resolve Conflict:** Additionally, the work has defined a method for detecting conflicts between refactorings. The fact that the detection is at the more fine-grained FGT-level as opposed to refactoring-level, means that the source of conflicts can be accurately pin-pointed and resolved by manipulating FGTs rather than refactorings.

5. **Find Sequential Dependency:** A method for finding the sequential dependency that may occur between refactorings has defined in the work. To do that, the method is also based on the idea of finding the sequential dependency at the level of FGTs.

6. The concept of a **"refactoring deadlock"** has analysed, and a method to detect a deadlock between two refactorings has been proposed.

7. Conditions under which **"ambiguity"** in the sequential dependency between two refactorings arises, has been identified and catalogued. A method to solve such ambiguity has been proposed.

8. **Composite Refactorings:** The work has introduced a methodology to deal with composite refactorings in an FGT context. The methodology constructs the composite refactoring from a collection of FGTs with a set of composite-level pre- and postcondition conjuncts. Because the resulting composite is expressed in terms of FGTs, the composite can be analysed with respect to conflict, redundancy, sequential dependency and parallelizing opportunities—just as any other FGT-based refactoring. Furthermore, by suitably checking preconditions against an existing system, rollback can be avoided—just as in the case of previous approaches.

8. **Parallelizing Opportunity:** The work naturally exposes parallelizing opportunities at the time of refactoring or during the process of detecting conflicts, removing redundancies and finding sequential dependencies between refactorings. This is basically because the FGTs for a refactoring are classified into FGT-DAGs, depending on the sequential dependency between these FGTs. These FGT-DAGs are independent and can be managed concurrently.

9. **New Refactorings:** The work has established the basic foundation for giving the end users the ability to create new refactorings whose semantics is constrained, not by the selection of existing refactorings that have been implemented in the tool, but rather by the semantics of

the FGTs that have been predefined in the tool. This can be based on a DSL that can be used to create a more complex refactorings.

The differences between refactoring based on an FGT paradigm and those of alternative approaches are summarized in Table 13.1.

Table 13.1: A comparison between FGTs-based and alternative formalisms

Alternative Formalisms	FGTs-Based Formalism
→ Refactoring is a black box.	→ Refactoring is a collection of FGTs ordered in one or more FGT-DAGs.
→ Refactoring precondition conjuncts are defined at one level. (The same for postcondition conjuncts)	→ Refactoring precondition conjuncts are defined at two different levels. (The same for postcondition conjuncts)
→ No possibility of knowing which part of refactoring causes the conflicts. Therefore, it is difficult to resolve these conflicts.	→ Conflicts are detected at the level of FGTs. These conflicts can be resolved.
→ Less parallelizing opportunities.	→ More parallelizing opportunities.
→ Difficult for end users to build their own refactorings because there is a need to write a code.	→ Building new refactorings can be done by using the list of the proposed FGTs without a need to write a code.
→ Redundancy can only be removed at a refactoring-level.	→ Redundancy between FGTs can be removed.
→ No possibility to know at what specific point or points two refactorings are sequentially dependent.	→ Ability to know at what point or points two refactorings are sequentially dependent.

In general, there will be more FGTs than there are refactorings, and therefore more computational operations. However, this additional computational cost buys more flexibility—including, and especially, the flexibility afforded to the end user to define a wider range of refactorings than is possible when relying on primitive refactorings as building blocks. In the contemporary world of high-speed processors, and the relatively small scale of entities to be processed in a design level refactoring applications (typically in the order of thousands rather than millions or billions) the additional processing cost does not seem to be a significant factor. Moreover, it should also be borne in mind that the additional computational cost can be offset against the enhanced scope for parallelizing operations afforded by the FGT paradigm, where one can rely on the ever-increasing number of multi-core processors available on contemporary chips.

The next section considers further directions that this work could take.

13.3 Future Work

There are a variety of future challenges that require further investigation. Each subsection below contains a list of projects that could be undertaken by future researchers.

1. UML Meta-Model Extension: The work in the thesis is based on the simplified UML meta-model shown in Figure 1.5. For a full, mature and ready-to-use refactoring tool, an extension of the meta-model is needed to deal with constructs such as interfaces, aggregations, constructors and so on. It is to be expected that new dependencies and conflicts between the different FGTs will be introduced, and ways will have to be found deal with these.

2. Different Types of Software Artifacts: The discussion of the proposed approach in our work was based on applying refactorings to UML class diagrams. It may be possible to extend the approach to a wider range of UML modeling notations such as state and sequence diagrams. It may also be possible to extend the approach to the code-level, to database schemas, to software architectures or to the software requirements' levels. A more thorough investigation into these possibilities is needed, both in terms of feasibility and in terms of desirability.

3. Consistency: Throughout this work, the refactorings are reflected at the UML class diagram level. Ideally, the modifications should also be reflected on the other UML models affected by the refactoring, as well as on the code-level implementation of the system. This is because it is important to keep the different system models and code consistent with one another. Clearly, further research in this direction would be beneficial.

4. Removable-Conflicts: In chapter 8, three different kinds of conflicts between pairs of refactorings were described and treated: ordering-conflicts; cancelling-conflicts; and removable-conflicts. The resolving procedure for the first two kinds of conflicts is straightforward as discussed in chapter 8. Resolving the third kind of conflicts (removable-conflicts) need more attention. It is feasible to identify FGT pairs in the two refactorings that constitute removable-conflicts and it is also possible to offer guidelines about how one of the FGTs in the pair may be changed. However, there is no guarantee that resolving a removable-conflict will not introduce other conflicts. More investigation is needed on how to deal with detected removable-conflicts.

5. Deadlock Algorithm Extension: The deadlock algorithm presented in section 9.5 is concerned with finding a deadlock between two refactorings. However, it does not deal with

the fact that a deadlock may arise from circular sequential dependency relationships. For example, the following scenario leads to a deadlock:

$$A \rightarrow B,\ B \rightarrow C,\ C \rightarrow D,\ D \rightarrow A$$

A proper extension to the proposed deadlock algorithm should be investigated to deal with such cases, taking into consideration the algorithmic efficiency in the proposed solution.

6. Parallelizing Algorithms: Chapter 11 discussed the different parallelizing opportunities the new approach can open. Actual implementation of parallel algorithms should be investigated.

7. Larger scale example: The scope for parallelization has been explained in chapter 11. However, in order to explore the potential benefits of parallelization, large scale real-life examples should be investigated.

8. Domain Specific Language (DSL): Research is needed into developing a fully-fledged DSL for end users to create their own refactorings. The proposed language will have the features such as the following:

- A set of fully implemented FGTs with their pre- and postcondition conjuncts. This set will be ready for the user to select and use to construct a refactoring.
- A set of language constructs like (for example, if-statements, for-loops, etc). The syntax of these constructs should be specifically tailored to accommodate the FGT paradigm, they should be intuitive, easy to use, and should be sufficiently expressive for the user to assemble the desired sequence of FGTs to represent the intended refactoring.
- Support structures for the user easily to formulate refactoring-level pre- and postconditions. In the Prolog prototype, these took the form of procedures such as existsObject(--), supclass(--), subclass(--), isReferenced(--), etc.

While the development of such a DSL together with an environment in which it can be used is a non-trivial task, it seems like worthwhile endeavour that will maximally uncover the benefits to be derived from refactoring based on the FGT paradigm.