Search-Based and Goal-Oriented Refactoring using Unfolding of Graph Transformation Systems

Submitted to the

Department of Computer Science,

University of Leicester, UK.

by Fawad Qayum

A Thesis Submitted for the Degree of

Doctor of Philosophy

January 2012

Abstract

To improve automation and traceability of search-based refactoring, in this thesis we propose a formulation of using graph transformation, where graphs represent object-oriented software architectures at the class level and rules describe refactoring operations. This formalisation allows us to make use of partial order semantics and an associated analysis technique, the approximated unfolding of graph transformation systems. In the unfolding we can identify dependencies and conflicts between refactoring steps leading to an implicit and therefore more scalable representation of the search space by sets of transformation steps equipped with relations of causality and conflict.

To implement search based refactoring we make use of the approximated unfolding of graph transformation systems. An optimisation algorithm based on the Ant Colony paradigm is used to explore the search space, aiming to find a sequence of refactoring steps that leads to the best design at a minimal cost.

Alternatively, we propose a more targeted approach, aiming at the removal of design flaws. The idea is that such sequences should be relevant to the removal of the flaw identified, i.e., contain only steps which are directly or indirectly contributes to the desired goal.

Acknowledgements

First and foremost I offer my sincerest gratitude to my supervisor, Professor Reiko Heckel, who has supported me thoughout my thesis with his patience and knowledge whilst allowing me the room to work in my own way. I attribute the level of my PhD degree to his encouragement and effort and without him this thesis, too, would not have been completed or written. One simply could not wish for a better or friendlier supervisor.

I would also like to thank my examiners, Professor Dirk Yassens, and Dr. Neil Walkingshaw, for many constructive suggestions which improved this thesis. Specifically, I would like to acknowledge and extend my gratitude to Barbara könig for her thorough and continuous help during this work.

I am extremely grateful to University of Malakand for funding this research under Overseas Faculty Development Programme by the Higher Education Commission of Pakistan

Finally, I would particularly like to thank my mother and my wife Anisa Fawad for their constant prayers encouragement and support.

Words fail me to express my appreciation to my uncle Dr.Nisar and her wife Carol, whose love, support and persistent confidence in me, has taken the load off my shoulder and made my life easy in England.

In addition, I would like to thank many other members of the Department

of Computer Science at the University of Leicester for their help, and for creating a friendly and welcoming environment for work.

Dedication

This thesis is dedicated to my beloved father, Prof. Adbdul Qayum Khan (Late) who taught me that the best kind of knowledge to have is that which is learned for its own sake. *God rest his soul in peace. Ameen!*

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Refactoring has emerged as a successful technique to enhance object-oriented software designs by a series of small, behaviour-preserving transformations [35]. However, due to the number of design choices and the complex dependencies and conflicts between them it is difficult to choose an optimal sequence of refactoring steps, maximising the quality of the resulting design while minimising the cost of the transformation. Even in a system with only 20 classes the situation becomes acute because existing tools offer only limited support for their automated application [61]. Therefore, search-based approaches have been suggested in order to provide automation in discovering appropriate refactoring sequences [77, 40]. The idea is to see the design

process as a combinatorial optimisation problem, attempting to derive the best solution (with respect to a given quality measure or *objective function*) from a given initial design [65].

Apart from optimising given designs, refactoring is also considered as an effective tool for addressing design flaws. One accepted methodology is to identify an occurrence of an antipattern or code smell [83] known to affect negatively the desired quality of the code, and to use refactoring to remove the occurrence. This can be supported by tools for detecting design flaws and executing refactorings. However, the task of planning the refactoring sequence required to achieve the removal of the flaw is left to the developer. Again, this is non-trivial due to complex dependencies and conflicts between individual steps [61].

1.2 Problem Statement

Two obvious problems are the automation of the task of software refactoring [66], and traceability, i.e., the ability on behalf of the developer to understand the changes suggested by the optimisation [40]. In particular, heavy modifications make it difficult to relate the improvement to the original design, so that developers will struggle to understand the new structure. We believe that both problems can be mitigated by exploiting the local nature

of refactoring operations, which affect only a certain part of the design while leaving the context unchanged. In terms of scalability, local operations permit the use of partial order models representing the behaviour of a system by a set of actions (refactoring steps) equipped with relations of causality and conflict. Such models provide an implicit representation of the states (designs) of the system as conflict-free subsets of actions closed under causal dependencies, which scales better than the explicit representation of reachable states. For traceability, causal dependencies provide a model of explanation of why certain steps are required to perform later steps, thus reducing the problem to understanding the benefits of the final steps in a sequence.

Likewise, there is also a potential problem with the approach of using heuristic search to discover refactoring sequences automatically [40, 77] which doesn't focus on removing a specific design flaw, but uses a global software metric to guide the search for a good refactoring. The result, therefore, depends on the ability of the metric to capture the desired combination of qualities, typically leading to complex formulas that are hard to understand and evaluate [49]. A detailed analysis of the (potentially complex) metrics may be required for developers to accept the outcome and relate it to the original design. Otherwise, the accumulated understanding of the previous design may be lost.

1.3 Contributions

The contributions of this thesis can be summarised as follow:

1. *Studying how the formulation of refactoring as a graph rewriting can be exploited for optimisation.* Thus, we propose the idea of a formulation of refactoring based on graph transformation and the Ant Colony Optimisation metaheuristic (ACO). The main goal is to provide automation for the process of selecting refactoring sequences based on quality metrics, and formulate this as combinatorial optimisation problem to utilise locality of graph transformation with the help of the ACO metaheuristic.

2. *The mechanism of how refactoring can exploit the unfolding technique from the theory of graph transformation.* The analysis of graph transformation systems aids in summarising all potential interactions among transformation steps which enables us to detect the dependencies/conflicts among the proposed refactorings in order to an apply optimal sequence of transformations.

3. In search-based refactoring, there remains the problem of understanding the end product and *rationale* of the transformation. In order to improve understandability, the refactoring sequences generated should be

relevant to a specified goal of removing a given design flaw, i.e., contain only steps required to achieve that goal. We propose a more targeted approach, aiming at the removal of design flaws within a search-based framework, allowing for automation of the planning of refactoring sequences where each step, directly or indirectly, contributes to the desired goal.

Publications

The work done in this thesis expands and generalise material published in several research publications.

- Initially, we published in [72], the idea of a local formulation of refactoring based on graph transformation and the Ant Colony Optimisation metaheuristic (ACO).
- The idea of analysing refactoring dependencies using approximated unfolding of graph transformations was published in [71].
- We published in [70], the problem statement and implementation idea of using a combination of graph transformation theory and ACO meta heuristic, aiming to improve performance/scalability and traceability/understandability of search-based refactoring.
- We presented a detailed overview of the approach in [73].

1.4 Road Map

The thesis is compiled of two main parts. The first two chapters present the context of the thesis, while the next five chapters present the main contributions followed by the case study. The last chapter concludes the thesis.

Chapter 2 introduces the theoretical background and techniques that our contributions are based on including graph and hyper graph transformation, the presentation of refactoring as graph transformation, the ACO meta heuristic and counter-example guided abstraction refinement.

Chapter 3 discusses the different search based refactoring approaches, illustrates the analysis of refactoring dependencies and talks about the model refactorings.

The next four chapters present the main contributions. The overall methodology is explained in the Chapter 4. It has been categorised into two parts, addressing different problems in search-based refactoring. These are Search-based Refactoring and Goal-oriented Refactoring. Chapter 5 illustrates how we analyse dependencies and conflicts by exploiting the unfolding of graph transformation system. Refactoring as an optimisation problem is addressed in Chapter 6 and Chapter 7 presents the verification of sequence produced in the approximated model.

In Chapter 8, we present the implementation tool chain involved in our

methodology. Chapter 9 consists of the case study and evaluation. We conclude our thesis and presents future work in the Chapter 10.

Chapter 2

Background

In this chapter, we provide an introduction to the theoretical background of our research. It begins with the general concepts of graph transformation in the Section 2.1. Section 2.2 explains the unfolding of graph grammars, followed by demonstrating the idea of Petri graph, approximating the behaviour of graph transformation system and the tool called AUGUR2. Section 2.3 describes the modelling of refactoring as graph transformation. The Ant Colony Optimization metaheuristic is explained in the Section 2.6. The chapter is concluded by reviewing the technique of Counter-Example Abstraction Refinement (CEGAR) in Section 2.7.

2.1 Graphs and Graph Transformation

Firstly, in this section we need to introduce the essential concepts of graph and hypergraph transformations.

Definition 2.1.1 *(**Graph and Graph Morphism** [31]) A graph consist of $G = (V, E, s, t)$, where V is set of nodes (vertices), E is a set of edges such that each edge e in E has a source and a target vertex $s(e)$ and $t(e)$ in V, respectively. The two functions $s, t : E \to V$ are specifying the source and target of an edge.*

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

Given graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for $i = 1, 2$, a graph morphism $f : G_1 \to G_2$, $f = (f_V, f_E)$ consists of two functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ that preserve source and target, i.e., with $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$:

$$\begin{array}{ccc} E_1 & \overset{s_1}{\underset{t_1}{\rightrightarrows}} & V_1 \\ \downarrow{\scriptstyle f_E} & = & \downarrow{\scriptstyle f_V} \\ E_2 & \overset{s_2}{\underset{t_2}{\rightrightarrows}} & V_2 \end{array}$$

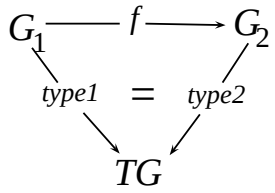
A graph morphism f is injective (or surjective) if both functions f_V, f_E are injective (or surjective, respectively); f is called isomorphic if it is bijective, which means both injective and surjective.

A type graph describes a set of types, which can assign a type to the nodes and edges of a graph. The typing itself is done by a graph morphism between the graph and the type graph.

Definition 2.1.2 *(**Typed Graph and Typed Graph Morphism** [31])*
A type graph is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ *where* V_{TG} *and* E_{TG} *are called the vertex and edge type alphabets, respectively.*

A tuple (G, type) of a graph G *together with a graph morphism* $type : G \to TG$ *is called a graph typed over* TG*.*

Given typed graphs $G_1^T = (G_1, type_1)$ *and* $G_2^T = (G_2, type_2)$*, a typed graph morphism* $f : G_1^T \to G_2^T$ *is a graph morphism* $f : G_1 \to G_2$ *such that* $type_2 \circ f = type_1$*.*



A production rule $p : L \to R$ consists of a pair of TG-typed instance graphs L and R. L represents the left-hand side graph (LHS) and R represents the right-hand side graph (RHS). Applying rule p to a source graph means finding a match of L in the graph and replacing it with R, thus creating the target graph.

In the DPO approach [32], a graph K is used as the common interface of L and R, usually their intersection. Hence, a rule is given by a span $p : L \leftarrow K \rightarrow R$.

Definition 2.1.3 *(**Graph Production** [22]) A (typed) graph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is composed of a production name p and two injective graph morphisms $l : K \rightarrow L$ and $r : K \rightarrow R$. L, K and R are (typed) graphs called left-hand side, gluing graph (or interface graph) and right-hand side respectively.*

A graph transformation with a production p is defined by first finding a match m of the left-hand side L in the current host graph G. Then all the vertices and edges which are matched by $L \setminus K$ are removed from G. An intermediate graph D is created by $D := (G \setminus m(L)) \cup m(K)$. It has to be a legal graph, i.e., no edges should be left dangling. Hence, the match m has to satisfy a suitable gluing condition, which makes sure that the gluing of $L \setminus K$ and D is equal to G (see (1) in Figure 2.1). In the second step of a graph transformation, the graph D is glued together with $R \setminus K$ to obtain the derived graph H (see (2) in Figure 2.1).

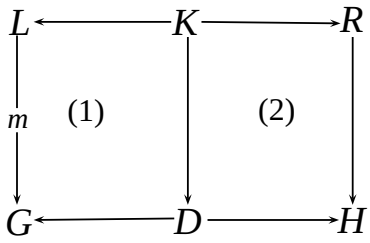


Figure 2.1: DPO graph transformation

Definition 2.1.4 *(**Graph Transformation** [22]) Given a (typed) graph production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *and a (typed) graph* G *with a (typed) graph morphism* $m : L \to G$*, called the match, a direct (typed) graph transformation* $t = G \overset{p(m)}{\Longrightarrow} H$ *to be constructed, producing a new graph* H *by construction the double pushout diagram in Figure 2.1.*

A graph transformation is a sequence of production applications. A set of production rules over a common type graph form a *graph transformation system* $\mathcal{R}$. A *graph grammar* $\mathcal{G}$ is a graph transformation system $\mathcal{R}$ with a start graph $\mathcal{G}_{\mathcal{R}}$.

Now we will introduce hypergraph transformation. The difference with the more common notions of typed or labelled graphs with binary edges is that, in hypergraphs, only edges are labelled and each edge can be connected to a finite sequence of nodes, rather than just one source and one target. Let Λ denote a fixed set of *labels* and each label $l \in \Lambda$ is associated with an *arity* $ar(l) \in \mathbb{N}$.

Definition 2.1.5 *(**Hypergraph**) According to [6] a (Λ-)hypergraph G is a tuple (V_G, E_G, c_G, l_G), where V_G and E_G are finite sets of nodes and edges respectively, $c_G : E_G \to {V_G}^*$ is a connection function and $l_G : E_G \to \Lambda$ is the labelling function for edges satisfying $ar(l_G(e)) = |c_G(e)|$. Nodes are not labelled.*

A node $v \in V_G$ is called isolated if it is not connected to any edge, i.e. if there are no edges $e \in E_G$ and $u, w \in {V_G}^*$ such that $c_G(e) = uvw$.

For example Figure 2.2 depicts a hypergraph with three edges labelled as $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and three nodes.

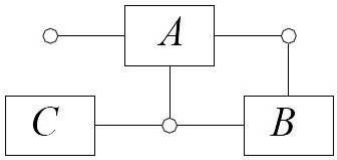


Figure 2.2: A hypergraph

Let G, G' be *hypergraphs*. A hypergraph morphism $\varphi : G \to G'$ consists of a pair of total functions $\langle \varphi_V : V_G \to V_{G'}, \varphi_E : E_G \to E_{G'} \rangle$ such that for every $e \in E_G$ it holds that $l_G(e) = l_{G'}(\varphi_E(e))$ and $\varphi_{V^*}(c_G(e)) = c_{G'}(\varphi_E(e))$.

Definition 2.1.6 *(**Hypergraph Rewriting Rule** [6]) A rewriting rule r is a triple (L, R, α) where L and R are hypergraphs, called left-hand side and right-hand side, respectively, and $\alpha : V_L \to V_R$ is an injective function*

mapping the nodes of the left-hand side to nodes of the right-hand side.

Intuitively, a rule $r = (L, R, \alpha)$ specifies that an occurrence of the left-hand side L can be replaced by R. To apply a rule r to a graph G at the match $\varphi : L \to G$, we first remove from G the images of the edges of L. Then the graph is extended by adding the new nodes in R i.e., the nodes in $V_R - \alpha(V_L)$ and edges of R. A rule can delete and produce but not preserve edges, while nodes cannot be deleted and its left-hand side must be connected [6].

A graph transformation system is a finite set of rewriting rules. A graph transformation system with a start graph is called a graph grammar $\mathcal{G}$.

Rules $r = (L, R, \alpha)$ can be seen as DPO rules $(L \hookleftarrow V_L \overset{r}{\hookrightarrow} R)$ [33] such that hypergraph rewriting is equivalent to a DPO construction. Figure 2.3 illustrate the idea of a rewriting rule and its corresponding DPO rule, where mapping α corresponds to the numbering of nodes from the left-hand side to the right-hand side. Nodes without numbers on the right-hand side are the result of the application of the rule [7].

2.2 Unfolding of Graph Grammars

Before we give an impression of the technique, presented in [6], for the construction of a finite approximation of the unfolding of a graph grammar, we summarise the class of (hyper)graph transformation systems explained in the

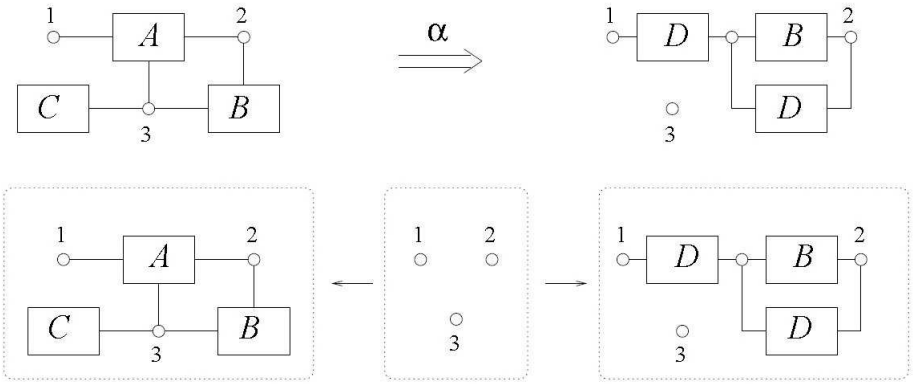


Figure 2.3: A hypergraph rewriting rule and its corresponding DPO rule

previous section. Then we go over the basic concepts underlying the ordinary unfolding construction for graph grammars [8]. Afterwards we demonstrate Petri graphs, the structure combining hypergraphs and Petri nets, which is used to approximate the behaviour of graph grammars.

The unfolding, an initially formulated for Petri nets [64], is based on the notion of associating a system to a single branching structure, describing all its possible runs, with all the possible events and their mutual dependencies. We are using this technique presented in [8], which unfolds a (hyper)graph transformation system. It starts with the initial hypergraph and produces a branching structure by applying at each graph in all possible ways the rules, without deleting the left-hand side. Consequently, it extracts an approximation called Petri graph, describing the behaviour of a graph grammar [6].

2.2.1 Petri Graph

First we explain Petri nets before describing the structure that we intend to use to approximate graph transformation systems, the so-called Petri graphs, because it consist of an hypergraph and of a Petri net whose places are the edges of the graph.

Definition 2.2.1.1 *(**Petri net** [6]) Let A be a finite set of action labels. An A-labelled Petri net is a tuple* N=*$(S, T, {}^\bullet(), ()^\bullet, P)$ where S is a set of places, T is a set of transitions, ${}^\bullet(), ()^\bullet : T \to S^\oplus$ assign to each transition its pre-set and post-set and $p : T \to A$ assigns an action labels to each transition.*

A marked Petri nets is pair (N, m_N), where N is a Petri nets and $m_N \in S^\oplus$ is the initail marking.

It is important to mention the relation between Petri nets and DPO graph grammars. As observed by Kreowski in [55], graph transformation systems can model the behaviour of Petri nets. The proposed encoding of nets into grammars represents the topological structure of a marked net as a graph, in such a way that the firing of transitions modelled by direct derivations [8].

A Petri graph is a static data structure, which records the transformation process. Combining a hypergraph and a Petri net, it is used to approximate a graph transformation system [6].

Let $\mathcal{R}$ be a graph transformation system. A Petri graph over $\mathcal{R}$ is tuple: *P=(G, N, μ)*, where G is a hypergraph, $N=(E_G, T_N, {}^\bullet(), ()^\bullet, P_N)$ is an $\mathcal{R}$-labelled Petri net, where the places are the edges of G, and μ associates to each transition $t \in T_N$, with $p_N(t) = (L, R, \alpha)$, a hypergraph morphism $\mu(t) : L \cup R \to G$ such that:

$$^\bullet t = \mu(t)^\oplus(E_L) \wedge t^\bullet = \mu(t)^\oplus(E_\mathcal{R})$$

This condition allows to interpret a transition in the net as an occurrence of a rule in $\mathcal{R}$.

A Petri graph for a graph grammar *(R, $G_\mathcal{R}$)* is a pair *(P, ι)*, where *P=(G, N, μ)* is Petri graph for $\mathcal{R}$ and $\iota : G_\mathcal{R} \to G$ is a graph morphism. The multiset $\iota^\oplus(E_{G_\mathcal{R}})$ is called initial marking of the Petri graph. Precisely, if $p_N(t) = (L, R, \alpha)$ and $\mu(t) : L \cup R \to G$ is the graph morphism associated to the transition, then $\mu(t)_{|L} : L \to G$ must be a match of r in G, i.e., the image of the edges of L in G coincides with the pre-set of t. The result of applying the rule to the considered match must be already in graph G, and the corresponding edges must coincide with the post-set of t. Petri graph consists of two components, i.e., a graph component that conveys structural information and a Petri net component that represents causality, conflicts and concurrency. It is important to note that the hyperedges of the graph component are at the same time the places of the Petri net.

2.2.2 Approximated Unfolding

Usually, the unfolding is infinite, even in the case of finite state systems. In order to ensure that the unfolding construction generates a finite structure, the authors in [6] propose to consider – besides the unfolding rule, which extends the graph by simulating the application of a rule – *a folding rule*, which, under suitable conditions, allows us to merge different parts of the structure that have been generated.

To achieve an *unfolding* step one has to find a match φ of a rule r, i.e., an occurrence of its left-hand side in the current graph. Afterwards the rule is applied but only adds new items as specified by the right-hand side of the rule without deleting the match (left-hand side). On the other hand, in a *folding step,* one has to find two matches of the same rule r. Then the two occurrences of the left-hand side of rule r in the current graph are merged. In both kinds of steps the matches are to be coverable, i.e., when interpreted as markings of the Petri graph they must be covered by a marking reachable from the initial marking in the Petri graph.

The algorithm used for the construction of the approximated unfolding of a graph grammar $\mathcal{G}$ is based on these two operations unfolding and folding, which can be illustrated as below [7]:

- *(Step 0)* $\rightarrow$ It begins with start graph of the grammar;

- *(Step i+1)* $\rightarrow$ Let $\mathcal{U}_i$ be the Petri graph produced at step i. Choose non-deterministically one of the following actions:

• **Folding**: Find a rule r and two matches of r coverable by a marking reachable from the initial marking in $\mathcal{U}_i$ such that one causally depends on the other, i.e., each item x in the second match causally depends on the transition labelled r applied to the first match or coincides with an item in the first match. Then merge these two matches.

• **Unfolding:** Find a rule r and a match of r coverable by a marking reachable from the initial marking in $\mathcal{U}_i$ such that the match cannot be involved in a folding step (and no other r-labelled transition has this match as precondition). Then apply the unfolding step to such a match.

These operations are applied subject to certain conditions and the algorithm stops when no (folding and unfolding) step can be performed [6]. The resulting Petri graph is called approximated unfolding of $\mathcal{G}$ and denoted by $\mathcal{A}(\mathcal{G})$. This guarantees that the resulting Petri graph is finite.

In order to demonstrate the idea of unfolding/folding steps in the approximated unfolding, we consider the graph grammar as an example from [7] representing a simple distributed system with mobility, with locations and processes running on these locations. Locations, connections and processes are characterized as hyperedges. The edge labels represent the following: P denotes a process running on a location, Q stands for a process, which trav-

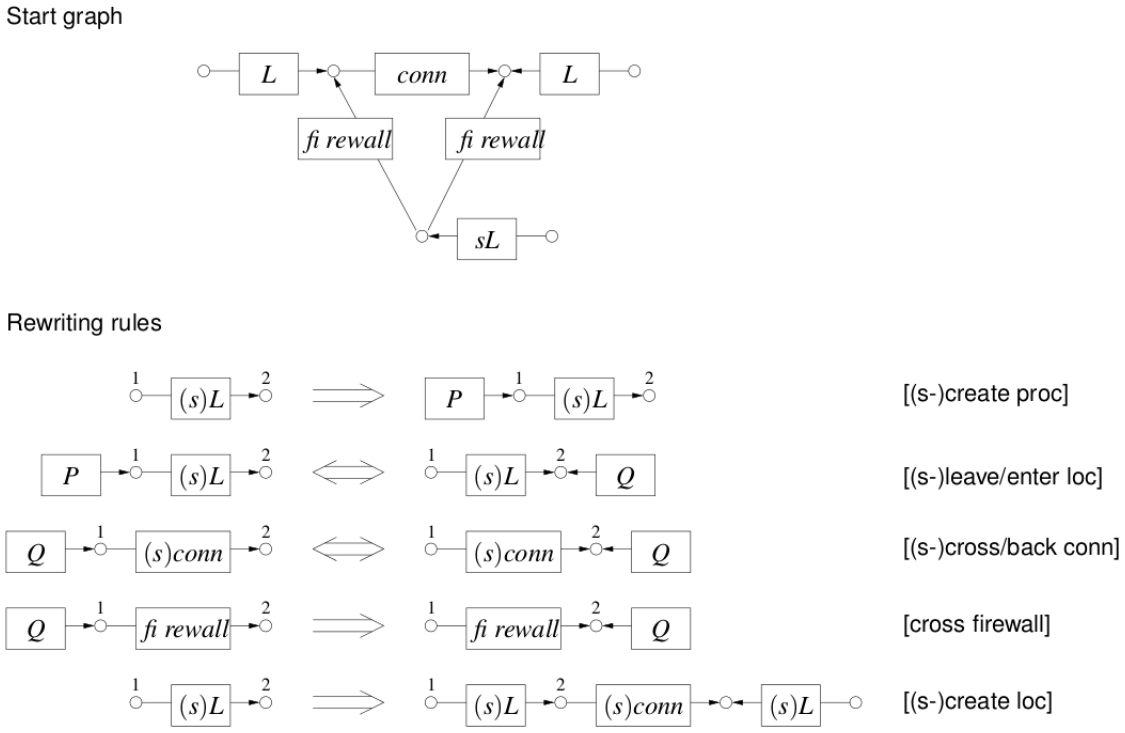


Figure 2.4: A graph grammar $\mathcal{M}$ for mobile processes.

els between locations. The labels L and sL stand for locations and secure locations, respectively, and, in the same way, *conn* and *sconn* stand for connections and secure connections, respectively. And hence *firewall* represents the one directional firewall connection.

For instance, the start graph in Figure 2.4 represents a network with three locations, among one of them is secure. The secure location is connected to others through firewalls and initially no process is running. The dynamics of the system is demonstrated with the help of rewriting rules depicted in the

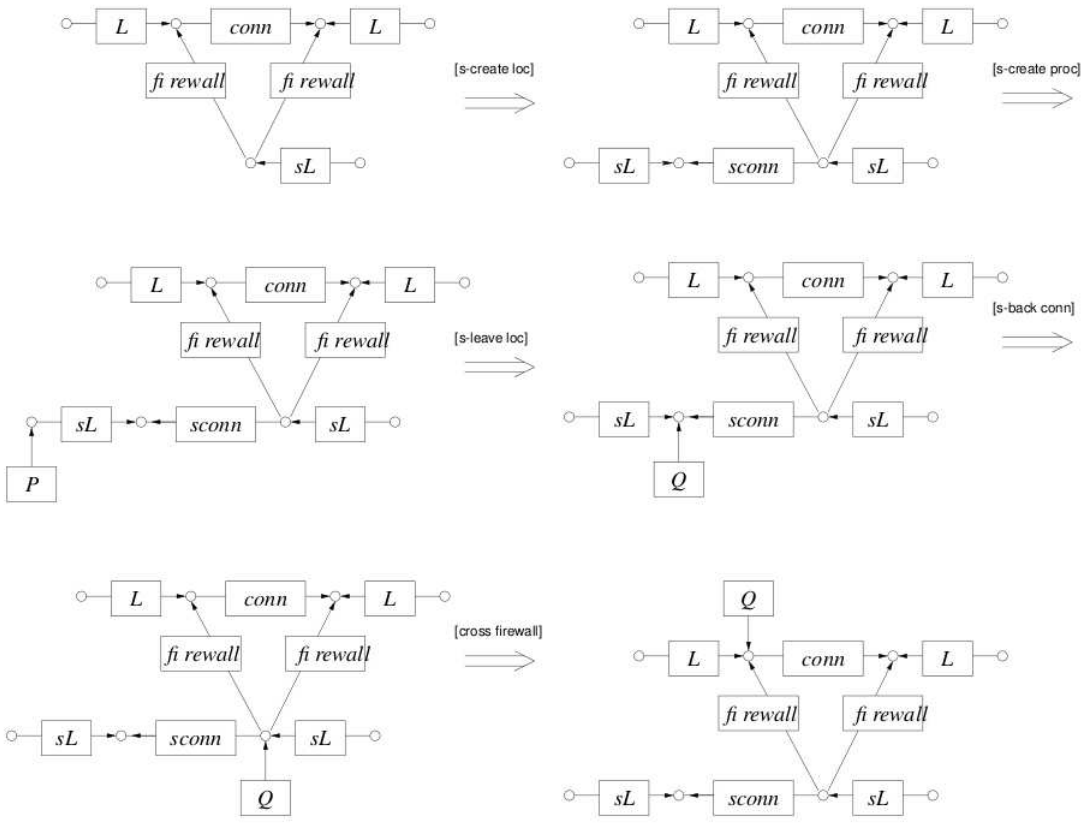


Figure 2.5: Few transformation steps in the mobile systems mention in graph grammar $\mathcal{M}$ of Figure 2.4.

Figure 2.4. Some of the rules can be applied in both directions, as indicated by the presence of a double arrow. A possible evolution of $\mathcal{M}$ is described in the Figure 2.5 i.e., a new secure location is created which later enables to create a new process. This process travels along the new secure connection and then crosses a firewall.

We exploit the algorithm to construct the approximated unfolding of the

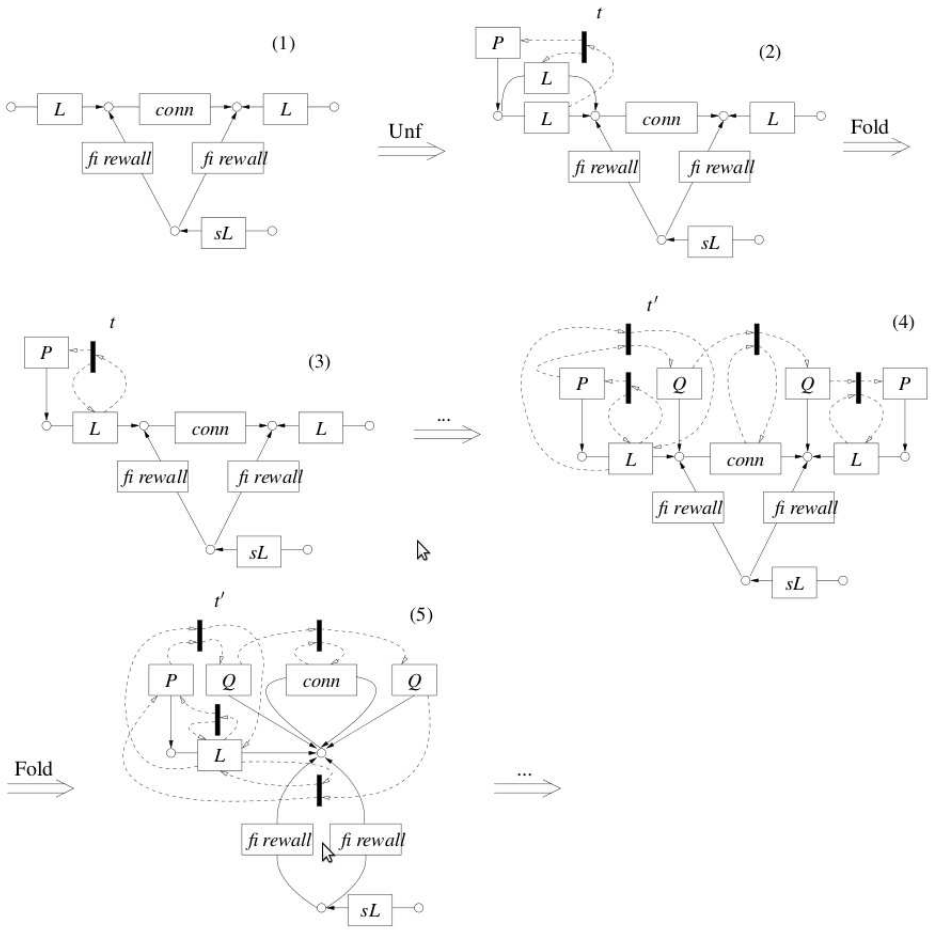


Figure 2.6: Folding and unfolding in the Approximated unfolding technique.

graph grammar $\mathcal{M}$ in Figure 2.4. It does several folding and unfolding steps and finally produces the Petri graph $\mathcal{A}(\mathcal{M})$ in Figure 2.7. A Petri graph contains both the graph structure of the system and a Petri net having edges as places. Both parts of the approximated unfolding are used to analyse the original system. For instance, the start graph of the grammar is represented by the marking of the Petri graph in Figure 2.7, consisting of the labelled

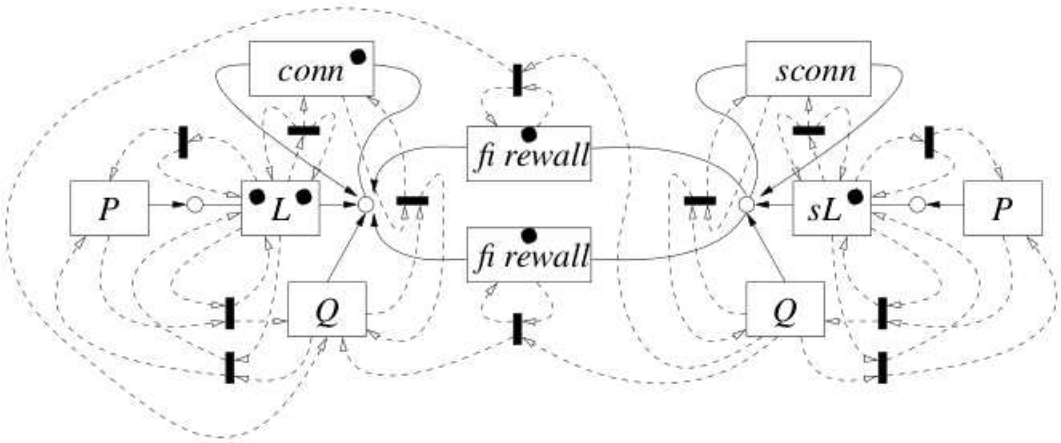


Figure 2.7: Approximated unfolding for the graph grammar $\mathcal{M}$ in Figure 2.4.

edges L, $conn$, $firewall$ and sL. The black rectangles represent transitions, which can be interpreted as occurrences of rewriting rules. Each transition t is labelled by the corresponding rule name (e.g., t: [create proc]) and it consumes and produces tokens, being the edges of the graph, as denoted by the dashed arrows.

Some of the unfolding and folding steps are presented in Figure 2.6: we start from the start graph of the system in Figure 2.4 and choose non-deterministically to apply an unfolding step corresponding to the creation of a new process at a non-secure location (rule [create proc]). A newly created location is a match for rule [create proc] and it causally dependends on transition t which corresponds to rule [create proc]. A folding step allows us to merge such an edge and the precondition of t. A series of several similar

unfolding and folding steps follows.

Similarly the process P and its location on the right represent a match for rule [leave loc] and they both causally depend on transition t', the match and the precondition of t' are consequently merged. Some more steps finally lead to the Petri graph in Figure 2.7.

2.2.3 AUGUR2

AUGUR2 is the verification tool based on the previously developed verification tool Augur, which can analyse graph transformation systems by approximating them by Petri nets [53]. As compared to the previous version, the new version AUGUR2, provides a more general and extensible software architecture and additional functionality for analysis and visualization. AUGUR2 provides new analysis techniques and more functionality. In particular, an analysis algorithm for Petri nets has been added, which is based on coverability graphs [75] and backward reachability [1]. Additionally, an interface to Graphviz[1] for visualization purposes has been set up and also added the possibility to specify forbidden paths in graphs using regular expressions [53].

In AUGUR2, rewriting takes place on hypergraphs, i.e., left and right-hand sides can be (almost) arbitrary hypergraphs. However, no nodes can be deleted and rules must be consuming, i.e., at least one edge is deleted.

[1] http://www.graphviz.org/

In order to provide input to AUGUR2, we need to translate the initial hypergraph and hypergraph rules into GXL (Graph Exchange Language), an XML standard for graph transformation systems [57]. AUGUR2 only allows GXL files as input for the purpose of constructing approximated unfolding. The output produced by AUGUR2 is in GXL, as well augmenting the Petri graph with information about nodes, transitions and extra tentacles connecting transitions with hyperedges [29]. This enables us to analyse the concurrent behaviour of the system including all possible transformation steps and the dependencies among them.

2.3 Modelling Refactoring as Graph Transformation

Graphs are often used as abstract representations of models. For example, in the UML specification [68] a collection of object graphs is defined by means of a metamodel as abstract syntax of UML models. Model transformations can be specified in terms of graph transformation. In order to provide a localised formal description of refactorings as input to the partial order analysis, we follow [61] representing refactoring operations as graph transformation rules. The graph we use to represent Java class structures is typed to a metamodel

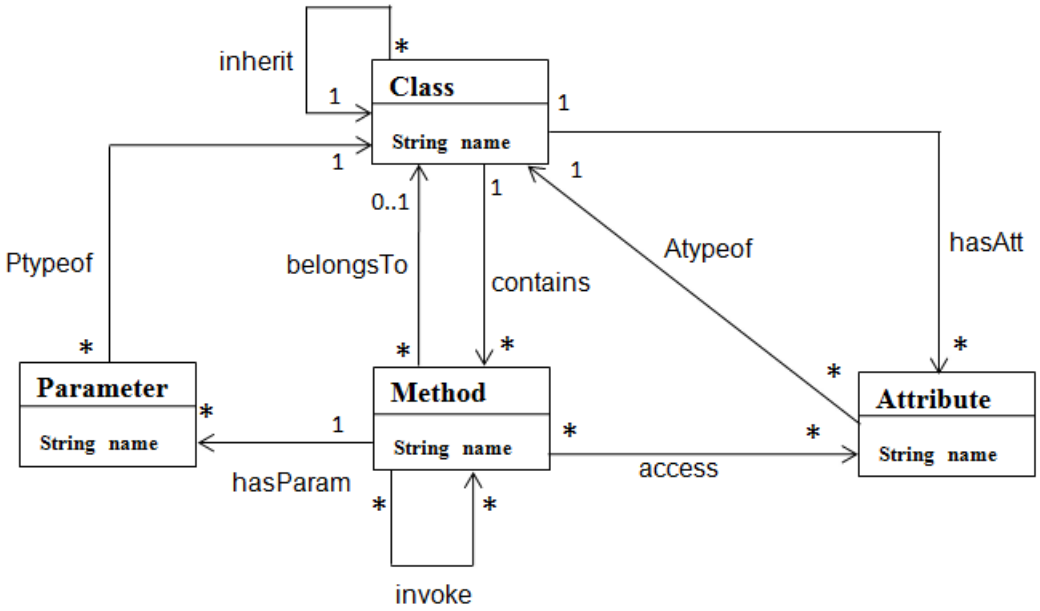


Figure 2.8: Meta model for representing object oriented Class Structure.

for representing object oriented class diagrams, and is depicted in the Figure 2.8. It shows the basic object oriented concepts like classes, methods, attributes and their relationships.

Class Level Refactoring

Refactoring has emerged as a successful technique to reduce the complexity of object oriented designs. We use graphs to represent software architectures at the class level and graph transformation to formally describe their refactoring operations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring.

Standard examples of refactorings are: moving a method from one class to another to better reflect that methods use; inserting a class into the inheritance hierarchy to capture common properties of subclasses, etc.

Below we present graph transformation rules for some of the class level refactorings [35], which will be considered later in this thesis.

Extract SuperClass: When two classes have similar features, we create a superclass for them and move their common features to this superclass. Graph transformation rules for the Extract Superclass refactoring are given for two different situations. In the first situation, if we have two classes with common features and they don't have a superclass, a superclass is created for them, while in the second situation, if the two classes with common features are in an existing inheritance hierarchy, a superclass is created for these two classes only and is inserted into the hierarchy. Both situations are depicted as Case A and Case B in Figure 2.9 respectively.

Move Method: When a method is using or is used by more features of another class than the class on which it is defined, we move that method to the other class. We consider three situations in order to move a method from one class to another. In the first case, we move a method to the class an invoked method belongs to. In the second case, we move a method to one of

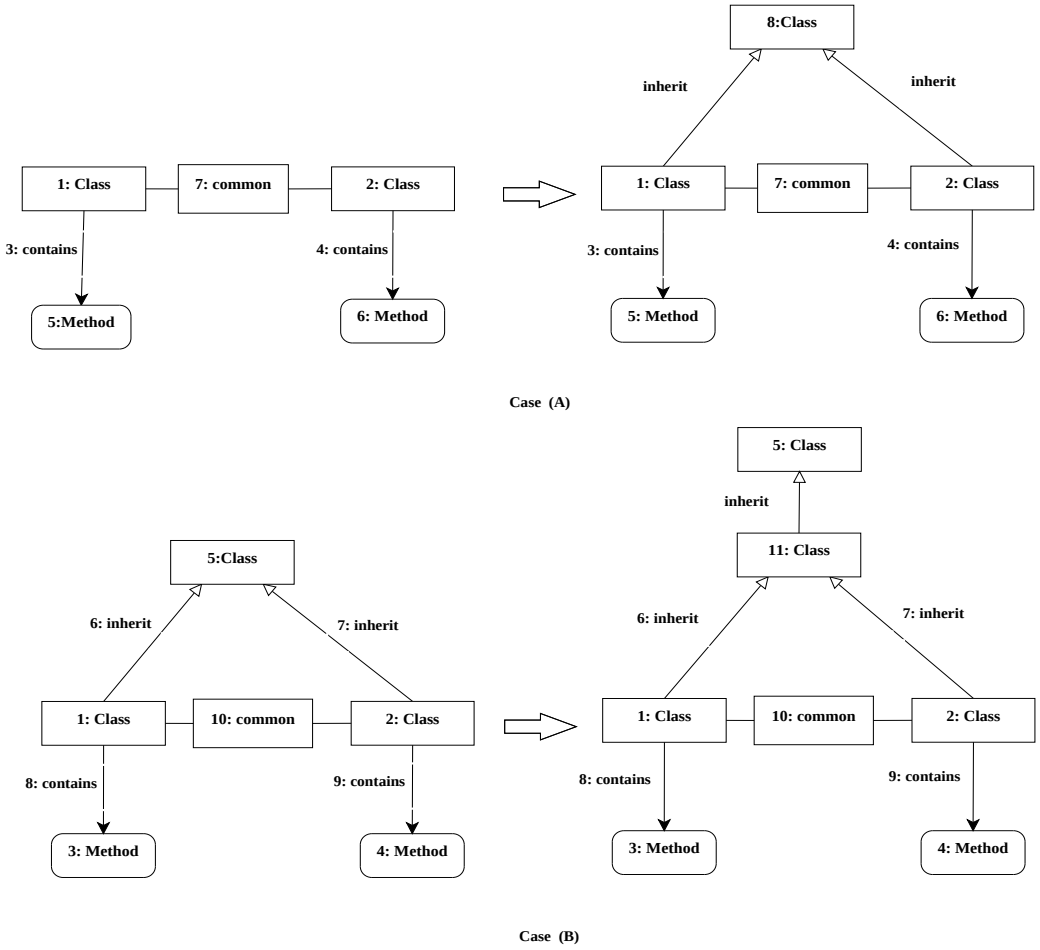
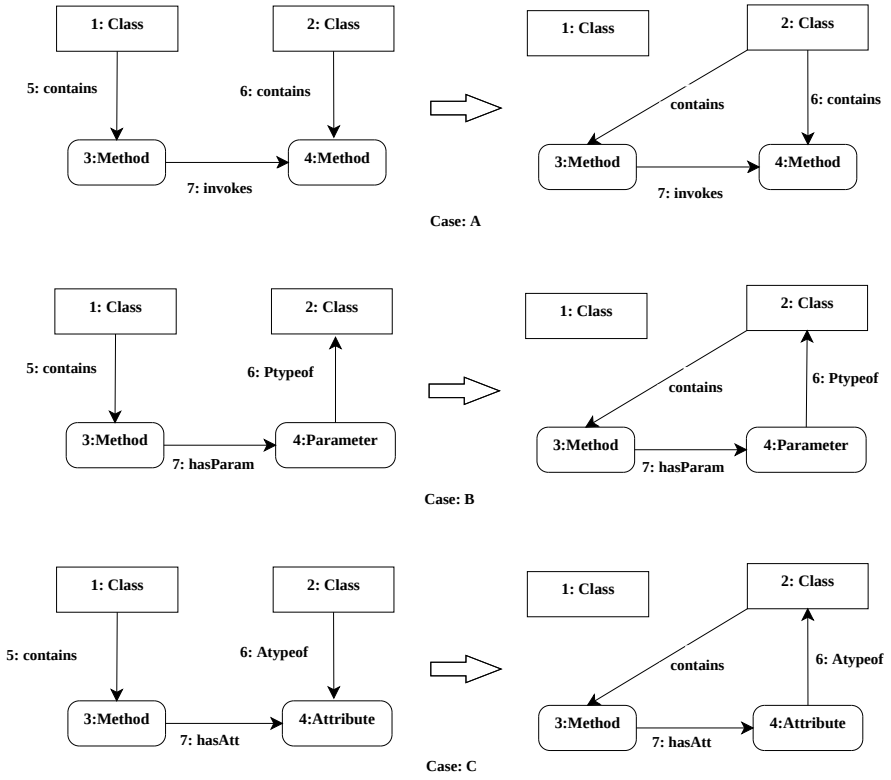


Figure 2.9: Graph Transformation rules for Extract Superclass refactoring

its parameter types. In the third case, a method is moved to the type of an attribute of the class. All three cases are shown in the following Figure.

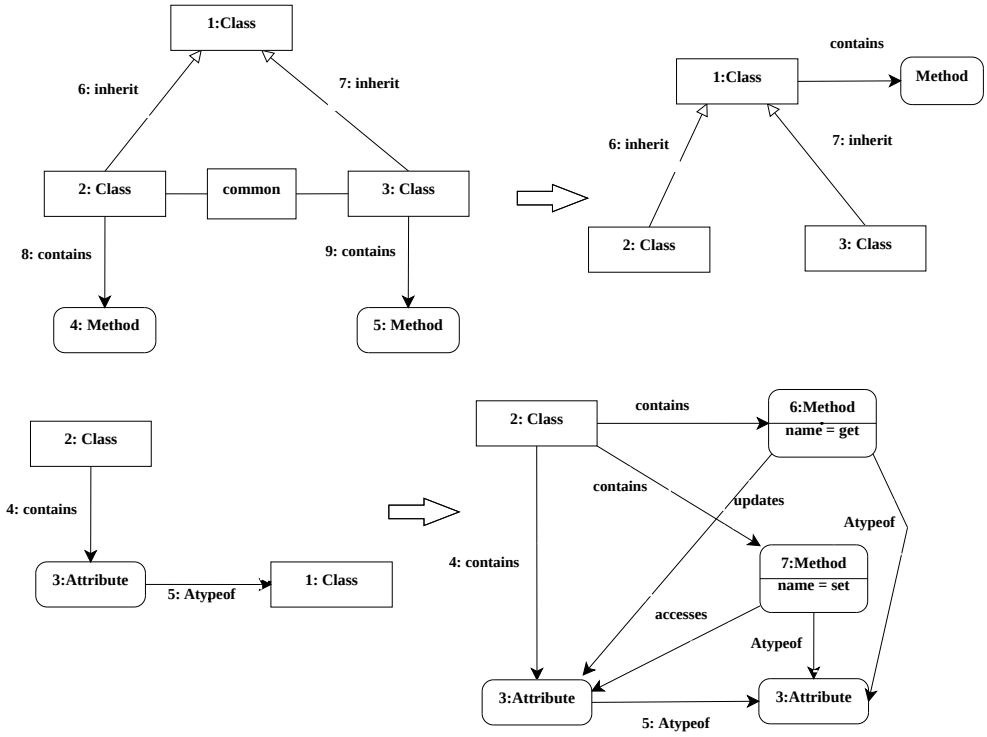
PullUp Method: When you have methods with identical results on subclasses, move them to the superclass. The graph transformation rules for the Pull Up Method refactoring is depicted below.



Encapsulate Attribute: When we need to increase modularity by avoiding direct access of the attribute, we change its visibility from public to private. The corresponding graph transformation rule for Encapsulate Attribute refactoring is given below.

2.4 Search Based Software Engineering

In recent years, research in search based software engineering (SBSE) has shown the potential of the approach. The idea is to see software engineering problems as combinatorial optimization problem or search problems [65]



utilising a large set of metaheuristic techniques such as genetic algorithms, simulated annealing and tabu search etc.

As asserted in [20], software engineers face problems, including not only of finding a solution, but ensuring that it is acceptable or near optimal solution from a large number of alternatives. Usually, it is difficult to reach an optimal solution, but more obvious to evaluate and compare candidates. E.g., it may be tough to know how to get a design with low coupling and high cohesion, however it is fairly easy to make a decision whether one design is more closely coupled than another. Generally, it is useful in problem domains with conflicting or competing goals, to anticipate many potential solutions

instead of a single perfect one, which is common in software engineering [42]. When any software engineering problem is designed as a search problem, a variety of approaches can be applied to that problem and allowing reuse of existing knowledge [66]. Therefore, Search-Based Software Engineering can be described as the application of search-based approaches to optimisation problems in software engineering [38].

A metaheuristic is a general algorithmic framework which is intended to find optimal or near optimal solutions to a problem having more than one solution in the search space [20]. Hence, metaheuristic algorithms have been applied successfully to variety of software engineering problems from requirements engineering [5], project planning and cost estimation [2, 3] through testing [11, 16, 39, 58], to automated maintenance [15, 34, 63, 67, 40, 77], service oriented software engineering [17], compiler optimization [21] and quality assessment [15].

2.5 Data Exchange Formats

In order to situate the background about the data exchange formats used later in this thesis, we provide a short description about MSE and GXL (Graph eXchange Language) data exchange formats.

2.5.1 GXL (Graph eXchange Language)

Research in reverse engineering and program analysis has established that there are plenty of tools available to extract information about a program for the purpose of manipulation and analysis. For the sake of interoperability among these tools, support for exchanging information was missing [46]. On the other hand, a lot of software tools depend on graphs for internal data representation, which offers support towards a standardized language for exchanging those graphs and improving interoperability between these tools [85]. Hence, the development of GXL (Graph eXchange Language) originally started to support data interoperability between reengineering tools.

GXL is an XML-based format for exchanging graphs [86]. Formally, GXL represents typed, attributed, directed, ordered graphs which are extended to represent hypergraphs and hierarchical graphs [45].

2.5.2 MSE

MSE is the generic file format used for import-export in Moose (a language-independent reengineering environment) [30]. Moose is open source software, which offers a comprehensive platform for software and data analysis. It provides a variety of services ranging from importing and parsing data, to modeling, measuring, querying, mining, and building interactive and visual

analysis tools. Different kinds of tools have been built around Moose. Some of these tools are distributed in the Moose Suite and others are to be loaded separately. Fame is one of these tools, and is a meta-modeling engine based on FM3 (Fame meta-metamodel) using the text-based exchange format (MSE).

Similar to XML, MSE is generic and can describe any model. MSE is part of the Fame [56] project and allows the specification of models for import and export with Fame. That means, MSE is a file format to store FM3 [56] compliant meta-models and models.

2.6 Ant Colony Optimization

ACO is applicable to a wide range of combinatorial optimisation problems. It is based on the metaphor of artificial ants cooperating to find a solution by searching a graph independently, but leaving pheromone deposits on the graph's edges to indicate ignore paths. To accomplish this, ants have to know the local neighbourhood of their current node, from which they will select the most likely edge to traverse based on the evaluation of the successor node and the pheromone value of the edge.

A metaheuristic is a general algorithmic framework which can be applied to different optimisation problems with relatively few modifications to make them adapted to a specific problem. - (Marco Dorigo and Thomas Stützle)

[28]

The capability of finding very high-quality solutions to hard, practically relevant combinatorial optimization problems in a reasonable time has considerably increased with the help of metaheuristics. This is particularly true for large and poorly understood problems [28].

Since the early nineties Ant Systems were suggested as a novel heuristic approach for the solution of combinatorial optimization problems [27]. Initially, they were applied to the traveling salesman problem, and along with necessary modifications to improve performance and apply to other optimization problems, the Ant Colony Optimization (ACO) metaheuristic emerged as an effort to describe a common framework for all the versions of AS [25].

The Ant Colony Optimisation (ACO) can be seenas a sophisticated optimisation strategy inspired by the foraging behaviour of ant colonies, and improved by artificial intelligence techniques [13]. This has been recognised as a strategy that can be used to solve complex optimisation problems. It is based on a set of artificial ants cooperating to find a solution by searching a graph independently, but leaving pheromone deposits on the graph's edges to indicate promising paths. Each single ant reveals a very minor behaviour: it simply goes from a node to another across an arc, but when all ants cooperate, like actual ants do in a real colony, the whole system reveals an intelligent behaviour, as much as it is able to find a good solution [26].

Artificial Ants

The ant agents used in the ACO metaheuristic are generally known as artificial ants. As compared to the natural counterpart, artificial ants are equipped with some extra capabilities to solve more complex real-world optimisation problems [28].

- **Visibility:** Artificial ants are given visibility when they come across intersections. With this ants are capable to judge the better path at the junction by using the probabilistic rules based on local information.
- **Memory:** Real ants are supposed have no memory and build paths on the basis of pheromone intensities of the decision path. On the contrary, memory is given to the artificial ants for storing records of previously visited paths.
- **Pheromone update:** Pheromone updating consist of two activities *pheromone evaporation* and *pheromone deposit*. In the first activity, pheromone evaporation reduces the value of the pheromone trails on all the arcs *(i, j)* by a constant value ρ, hence decreasing the difference in pheromone intensities among arcs. As a result, it can be considered as a source of encouraging exploration of unvisited paths by reducing the overall gap between pheromone trail intensities. During the

foraging process of real ants pheromone evaporation also takes place, but at much slower rate. In contrast, the higher evaporation rates are suggested for artificial ants, especially when solving more complex problems.

Deposition adds pheromone to the arcs as reward that it connects components contained in the candidate solution based on the objective function of the solution.

ACO for Combinatorial problems

Prior to applying any optimisation metaheuristic to solve a combinatorial optimisation problem, it is crucial that the problem can be represented in a form that is identifiable by the metaheuristic. Similarly, ACO is a metaheuristic widely applicable to any combinatorial optimisation problem for which a constructive solution procedure can be envisaged. To employ it, we need to map our problem into a representation allowing artificial ants to construct the solution by traversing a graph [28].

Problem representation

ACO assumes a problem representation with the following characteristics [28].

1. A finite set of solution components $C = \{c_1, c_2, \cdots, c_n\}$, and a set of arcs E connecting the components in C.

2. The states of the search problem, defined as sequences of components $x = \langle c_i, c_j, \cdots \rangle$ in C. The set of all possible states x is denoted X. The length (number of components) of a sequence is denoted by $\mid x \mid$.

3. A finite set S of candidate solutions with distinguished subset $\bar{S} \subseteq S$ of *feasible* candidate solutions determined by a set of constraints Ω.

4. A non-empty subset S^* of optimal solutions.

5. An evaluation $f(s)$ for each candidate solution s. For some problems it is possible to calculate *partial evaluations* $f_p(x)$ associated with intermediate states x of the problem.

Using the formulation above, artificial ants build solutions by performing randomised walks on the connected graph $G = (C, E)$, based on the following basic operations [26].

- A *state transitions* takes an ant from a one node to another across an arc;

- A *local update* changes the pheromone deposit on the arc it currently walks on;

- A *global update* changes the pheromone deposits on all arcs an ant has traversed when this ant successfully ends its trip;

In addition, we require a *comparison function* to evaluate different paths and an *end of activity* condition to specify when an ant has completed its trip.

2.7 Counterexample-Guided Abstraction Refinement

The techniques based on counterexample-guided abstraction refinement (CEGAR) have been used successfully for the purpose of verification [19]. The notion behind this approach is to start with a coarse initial over-approximation of a system and try to verify a certain property using this abstraction. If the property cannot be verified, there is a run, i.e., sequence of events in the Petri graph, that violates this property. This is known as counterexample [54]. The authors in [51] employed the technique of counterexample-guided abstraction refinement for the verification of graph transformation systems.

Their approach is based on approximating graph transformation systems by Petri nets by means of unfolding the construction [6]. The obtained approximation has the essential property that each graph reachable from the initial graph can be mapped, by combining some of its nodes, to a reachable marking of the over-approximating Petri net. Conversely, there might be some reachable markings in the over-approximation (Petri graph) having no

counterpart in the original GTS. Such sequence of events in the Petri graph is called a spurious run [51].

The reason behind spurious runs is the merging of graph nodes in the construction of the over-approximation, which is similar to the idea of summary nodes in shape analysis [76]. The construction of a more exact over-approximation proceeds by separating merged nodes in order to avoid this spurious run and the same process can be executed frequently for any number of spurious runs. This mechanism of finding the causes for spurious runs could be utilised in other frameworks based on approximations of graph structures [51]. It has been examined experimentally in [52] that counterexample-guided abstraction refinement is faster than already existing abstraction refinement technique mentioned in [10].

Let us consider the example from [51] illustrating a firewall system identical to the one presented in [7]. The system consists of two different types of processes, i.e., a safe processes (running behind a firewall) and unsafe processes (running in a public area). During runtime the system can generate any number of safe processes (SP) and connection locations (L). This is to make sure in the system that unsafe processes from the public area do not enter the firewall. This property needs to be verified but if such a situation is detected, then rule *Error* will be applied and an edge labelled Error is created. The initial graph and rules of the firewall system are depicted in

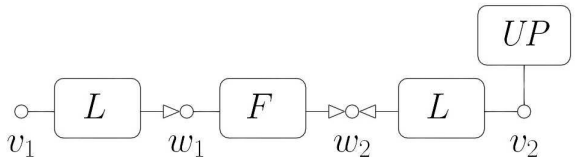


Figure 2.10: Initial graph of the firewall system

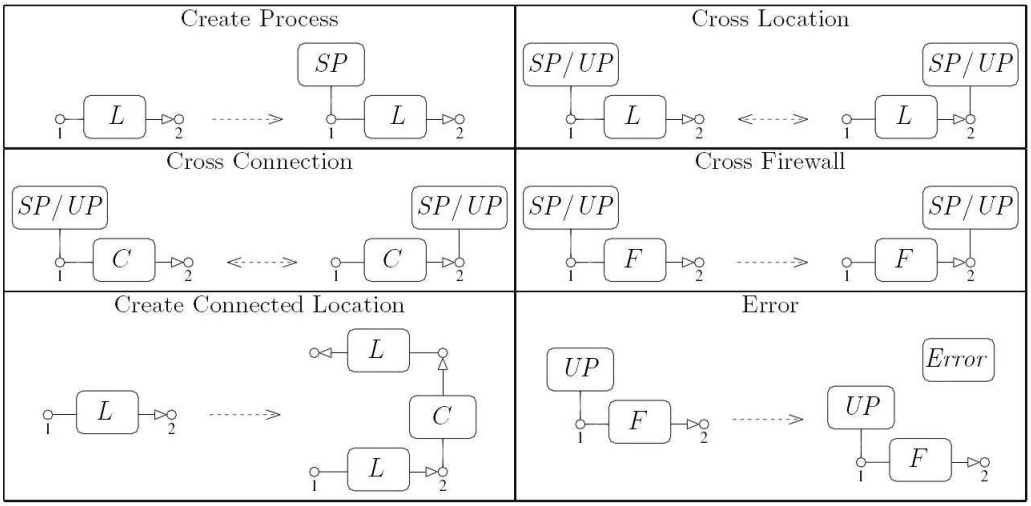


Figure 2.11: Rules of the firewall system

Figure 2.10 and 2.11 respectively. The rules with double-head arrow can be applicable in both direction.

After constructing the approximated unfolding of the graph grammar, which consists of the hypergraph in Figure 2.12 and the Petri net in Figure 2.13, the set of edges of the hypergraph corresponds precisely to the set of places of the net in the approximating unfolding.

In order to verify the graph grammar by analysing the Petri graph, we need to demonstrate that no reachable graph contains a subgraph G_s, which

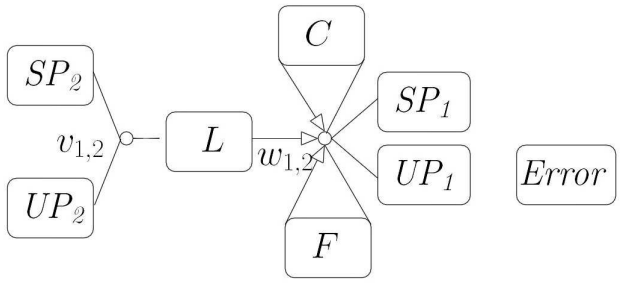


Figure 2.12: Hypergraph component of the underlying Petri graph

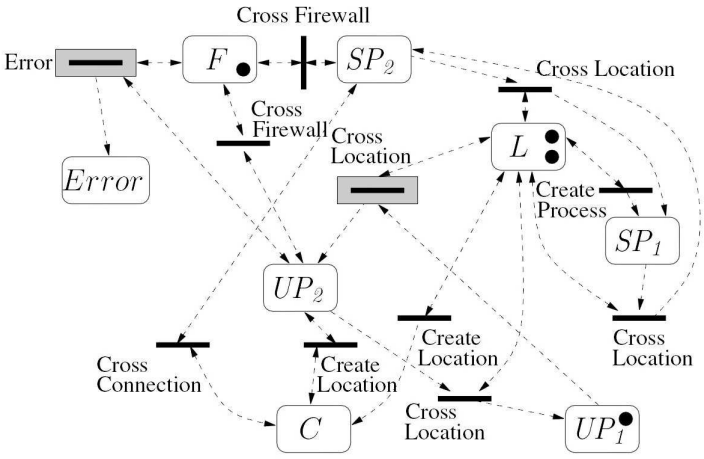


Figure 2.13: Petri net component of the underlying Petri graph

has been added to the grammar with G_s as left-hand side and an edge labelled *Error* in the right-hand (as rule *Error* in the Figure 2.11). The system will hold this property if and only if no place labelled Error is present in the underlying net or every such place is not coverable [51].

In the initial graph in Figure 2.10 and its hypergraph component in Figure 2.12 the nodes v_1 and v_2 as well as w_1 and w_2 have been merged by the over-approximation turned into $v_{1,2}$ and $w_{1,2}$. This is the case, which leads

to a spurious run.

Figure 2.14 illustrates the hypergraph component of the refined approximated unfolding obtained for the firewall system after the abstraction refinement. One can observe that crucial nodes of the hypergraph i.e., w_1 and w_2 are not merged.

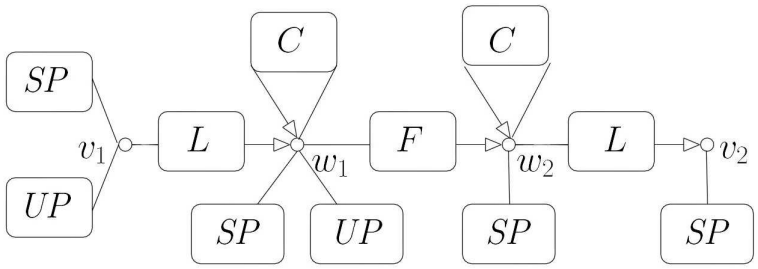


Figure 2.14: Hypergraph obtained after abstraction refinement

Chapter 3

Related Work

The purpose of this chapter is to clarify the contribution of this thesis in relation to other work in the area. Related work is described in three areas: search based software refactoring, analysis of refactoring dependencies and model refactorings.

3.1 Refactoring as Search Problem

Recently, the problem of refactoring has been addressed using search-based approaches [77]. The notion behind this is to formulate object-oriented design as a combinatorial optimization problem and the goal is to find an optimal design for a given initial model and objective function [65]. Having presented the refactoring task in this way, a variety of search techniques from

local searches, such as exhaustive search and hill-climbing, to metaheuristic searches, such as genetic algorithms and ant colony optimization, can be applied to solve the optimisation problem [66]. Since search-based refactoring has been introduced recently, it is not yet clear which search techniques are best suited in general.

Our approach analyses refactoring operations on the basis of graph transformation techniques, i.e., unfolding of graph transformation systems to identify conflicts and dependencies. Due to unfolding, our methodology doesn't express the state space explicitly but provides implicit representations of graphs as states.

Search-based refactoring often is based on a simple and small set of refactoring rules chosen from Fowler's catalogue [35]. E.g., in [41] they have built a general-purpose search based refactoring system, where they consider only the Move Method refactoring and authors in [65] had applied only two basic refactorings (PullUp Method and PushDown Method) in their approach. Instead, the prototype called CODe-Imp [67] uses six different refactorings, namely Push Down Field, Pull Up Field, Push Down Method, Pull Up Method, Extract Hierarchy and Collapse Hierarchy, but these refactorings are implemented individually for the purpose of design improvement. Likewise [77], implemented refactorings only for Move Method in order to improve the class structure of a system.

In our approach we consider a complex set of refactoring rules with dependencies and conflicts. This allows us to address a variety of design flaws at various levels within a system.

Below, we outline some of the approaches in the field of search-based refactoring in order to compare them to our contribution.

Initially, the idea to consider object-oriented design as a combinatorial optimization problem that can be solved with the help of search-based approach was coined by the O'Keeffe *et al.* in [65]. The authors have developed a prototype tool (Dearthóir) to confirm the concept of automated design improvement using simulated annealing. It takes Java code as input and chooses the task of moving methods to their best possible positions in the class hierarchy. In their proposed approach, they consider PullUp Method and PushDown Method refactoring [35] in a stochastic manner in order to improve the design of a system. Their only goal is to provide assistance to move methods to their most favourable positions in the class hierarchy, which shows that the approach is confined to a specific level in the system and supports a very limited number of refactorings.

Since the authors have presented their initial idea of automated design improvement, in [67] they have reformulated the refactoring task as a search problem in order to discover the extent to which software maintenance can be automated. They considered a small set of automatable refactorings to

find the appropriateness of the quality evaluation and assess the relative performance of a small set of search techniques in the perspective of automated refactoring of Java programs. But still their chosen set of refactorings addressed the design at the method/field level of granularity. The prototype (CODe-Imp) was aimed at automated designed improvement for search-based maintenance. Hence, in contrast to previous automated refactoring work, the authors have employed a more complex evaluation function in order to accommodate various metrics values. We believe that our approach is similar to this work in that the application of refactorings can provide a search space of alternative designs.

Similarly Seng *et al.* [77] described a search-based refactoring approach based on an evolutionary algorithm that suggests a list of refactorings in order to improve the class structure of a system. They transformed source code into an appropriate model with the help of fact extraction techniques in order to replicate the source code refactorings and find out their impact on a system. They classify the essential model elements, i.e., classes, methods, attributes, parameters and local variables in order to distinguish them as per their role in the system's design. The main similar idea between our work and this approach is extracting models from source code at the class level of granularity for the purpose of design improvement. In general, this approach has provided a valid bases for further research in the field of search-based

software engineering.

Followed the idea in [67], Harman and Tratt have employed the concept of Pareto optimality in search-based refactoring [41], allowing to combine different optimisation criteria into a single optimisation problem. The approach was helpful to resolve issues regarding evaluation functions, but so far the implementation of the approach is limited to the Move Method refactoring. In this approach the authors described the notion of "direct" and "indirect" approaches to search based refactoring and illustrated how existing search based refactoring approaches rely on complex fitness functions. We adopted this approach in terms of being indirect in nature because we are also optimising the sequence of refactorings and our evaluation function consist of different probe rules representing different optimisation criteria.

The authors in [14] have developed a tool called SORMASA, which identifies refactoring opportunities for improving the quality of software systems. SORMASA is using a search-based approach based upon a simple Genetic Algorithm in order to suggest refactoring actions (field and method level) to the user.

In this thesis along with finding an optimal sequence of refactorings according to some general quality metric, we also employ a more goal oriented search. Particularly, we are interested to suggest relevant refactoring sequences allowing the developer to address a specific design flaw. For in-

stance, if an improvement is witnessed by the programmer in the last step of the sequence, they will implicitly accept the relevance of all changes up to that point that the final step depends on.

Refactoring to patterns proposed that an existing design can be better improved with the help of patterns as compared to using patterns early in a new design [50]. Similarly, patterns allow us to find the locations in order to improve a design by applying pattern-directed refactoring. The authors mentioned in [48], *Design Patterns$\cdots$provide targets for your refactorings.* It means that there is an affiliation between patterns and refactorings because patterns show where to go and refactoring tells how to get there. We presume that our work is in some way similar to [50] in a sense that we have patterns and with the help of sequences of rules we achieve them. Our approach is not implementing the pattern but potentially we are refactoring to get rid of anti-patterns.

3.2 Analysis of Refactoring Dependencies

The authors in [61] presented refactorings as graph transformations, and proposed the technique of critical pair analysis to identify implicit dependencies between refactorings. Hence, the main objective of their work is to discover automated techniques in order to find out the implicit dependencies between

a list of refactorings.

To attain the above goal, they rely on graph transformation theory for a precise formal specification of refactorings and also they need to be capable of analysing mutual exclusion and sequential dependencies between refactorings. They employed the critical pair analysis [43] and sequential dependency analysis to identify mutual exclusions, asymmetric conflicts and sequential dependencies between refactorings respectively. In principal, critical pair analysis is useful to demonstrate conflicts, but it computes all potential conflicts. In a concrete situation, only a small part these conflicts may arise.

Our approach combines a formalisation of refactoring based on graph transformation [61] with the analysis of dependencies extracted from their approximated unfolding [53] by means of an optimisation using the Ant Colony-meta heuristic [28]. In [61] some of the potential conflict situations were not detected because of their specifications of refactoring were not complete. Sometimes more than one rule would be involved in the complete specification of refactorings. In contrast to [61] this thesis is based on the analysis of dependencies between actual refactoring steps as opposed to potential dependencies detected at the level of rules. In essence this is a “dynamic” approach, taking into account the given design as the actual input to the problem, as opposed to a “static” one.

3.3 Model Refactoring

The research on refactoring at model level is still in a developing stage and a number of problems are worthwhile for further research. Tool support for model refactoring is very limited. Often it is not even clear, what are appropriate refactorings on different types of models. An important question is, how model refactoring differs from program refactoring. Can notions, techniques and tools used for program refactoring also be applicable at the model level [60]?

In order to comprehend and investigate model refactoring a variety of formalisms have been presented in the literature. In [82], the authors used the formalism of Description Logic to detect behaviour inconsistencies during model refactorings. In another approach a forward-chaining logic reasoning engine has been used to support composite model refactorings [81]. In [37] the authors formalise a static semantics for Alloy, a formal object-oriented modelling language, in order to specify model refactorings. It is evident from the literature that UML models are regarded as suitable candidates for model refactoring. For example [78] presents a set of refactorings and describes how they can be designed in order to preserve the behaviour of a UML model. Similarly in [4], the authors proposed UML as a helping tool to find design smells and perform model refactorings. There are several

model transformation approaches based on graph transformation theory for the specifying of model refactorings, and rely on these formal properties to reason about and analyse these refactorings [12, 59, 61].

Our approach is using a similar formalism but addressing a different problem in the domain of refactoring, i.e., automation and traceability in search-based refactoring. Our research is originally inspired by the work of Mens *et al.* in [61]. Our methodology involves a local formulation of refactoring based on graph transformation. This allows us to make use of partial order semantics and an associated analysis technique, the approximated unfolding of graph transformation systems in order to identify dependencies between refactoring operations.

Several challenges in model refactoring have been discussed by the authors in [60]. For instance to provide a precise definition of model quality [80, 62], maintain consistency between model and program refactoring [84], behaviour preservation [74], testing model of refactorings [24], domain specific modelling [87] and analysing model refactorings [61]. Our work is an effort to address one of these challenges, i.e, analysing model refactorings. While in [61] they employ critical pair analysis techniques to analyse dependencies between refactorings, we propose the approximated unfolding construction for the analysis of refactoring operations.

Chapter 4

Methodology

In this chapter, we distinguish two approaches, addressing two different problems in search-based refactoring: *automation* and *traceability*.

We refer to them as:

- Search-based refactoring
- Goal-oriented refactoring

4.1 Overview

Both use a representation of object-oriented designs as graphs and of refactoring operations as graph transformation rules [61]. Such rules provide a local description, identifying and changing a specific part of the design graph only. After suitably encoding our rules into a hypergraph representation,

this enables us to derive a partial order structure of causality and conflict relations using the approximated unfolding of a graph transformation system [8] and its implementation in AUGUR2 [53]. The result is a structure called *Petri graph* [6]. Causal dependencies and conflicts, derived directly from the Petri graph, serve as input to our search problem.

Formally, our rules are based on hypergraphs representing instances of a metamodel for object-oriented software [70]. The unfolding of a graph grammar (a set of rules $p_1, \ldots, p_n$ modelling refactoring operations plus an initial graph G_0 representing the given design) starts from the initial (hyper)graph and applies all rules at all possible matches to the given and all resulting graphs. Rather than representing these graphs explicitly, only the rules and their matches are recorded. This leads to a compact representation of the behaviour of the system, further reduced by a scalable approximation obtained by folding potentially unbounded iterations. The result is an over approximation of the actual behaviour, potentially allowing for spurious transformation sequences. These have to be verified in the real model, possibly leading to a refinement of the approximation using the Counter-example Guided Abstraction Refinement (CEGAR) technique [51] .

4.1.1 Search-based Refactoring

To implement search-based refactoring we make use of the approximated unfolding of graph transformation systems. In the unfolding we can identify dependencies and conflicts between refactoring steps, leading to an implicit and therefore more scalable representation of the search space. An optimisation algorithm based on the Ant Colony paradigm is used to explore this search space, aiming to find a sequence of refactoring steps that lead to the best design at a minimal costs.

We employ Ant Colony Optimisation (ACO) [28] metaheuristic search for a solution. ACO is inspired by the behaviour of foraging ants, which search for food individually and concurrently, but share information about food sources and paths leading towards them by leaving pheromone trails. This amounts to a distributed traversal of a graph whose paths represent possible solutions [26]. In our case, the nodes of that graph could be the designs to be explored and its edges are the refactoring steps. Rather than representing this so-called construction graph explicitly, its nodes and edges are derived from the partial order structure as and when required. The ACO metaheuristic relies on an explicit representation of the search space. Thus, solutions and their local neighbourhoods have to be reconstructed on the fly from the partial order representation. The desired result is a

sequence of transformations leading from the given design to a design of high(er) quality, using only transformation steps that are optimal to achieve that improvement. As a result, a path (refactoring sequence) is produced representing the cheapest way to transform the given design into an optimal one.

A more detailed view of the approach is given by the diagram in Figure 4.1. Using UML activity diagram notation, boxes represent artifacts while oval nodes are the actions or transformations performed on them. For the sake of clarity, we divide the approach into three phases depicting different actions performed in an ascending order during the process.

Refactoring rules: Graph transformation rules formalising the refactoring operations are chosen from the standard catalogue [35] shared across all Java programs.

Encoding: A given Java program is translated into a graph representation.

Probe rules: We define probe rules in order to recognise patterns that are desirable or to be avoided in object-oriented designs.

Unfolding: We derive a partial order structure of causality and conflict relations, using the approximated unfolding of a graph transformation system [8].

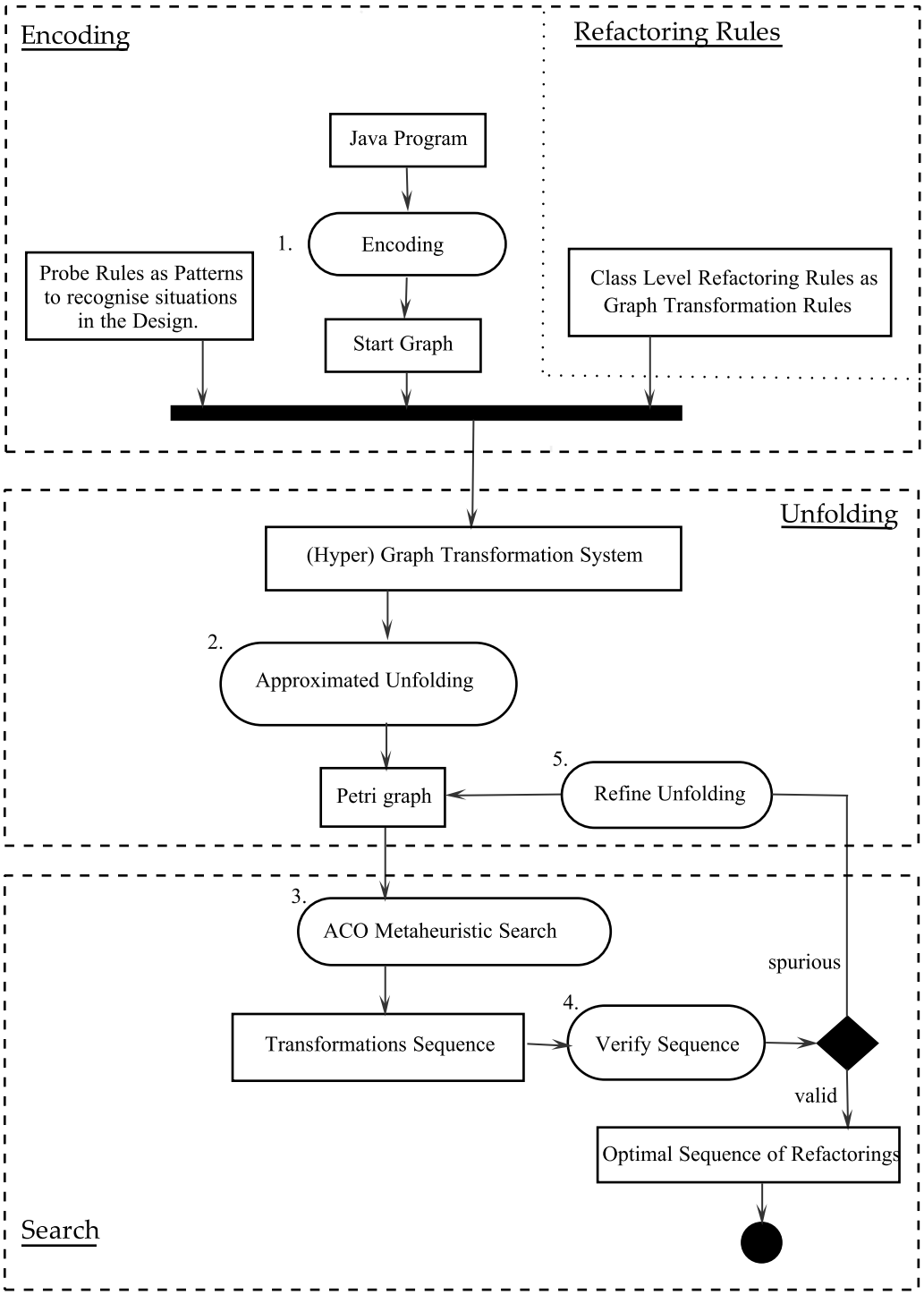


Figure 4.1: Abstract view of the Search-based Refactoring Approach

Search: The automatic refactoring problem is represented as an instance of the ACO metaheuristic. The partial order structure from the unfolding phase serves as input to the search algorithm.

Verification: If the optimal sequence doesn't exist in the real model and the sequence is rejected as spurious, then a refinement of the abstraction will be required leading to more accurate unfolding and another round of optimisation.

4.1.2 Goal-oriented Refactoring

Search-based refactoring is emerging as an approach to automate the improvement of object-oriented software. However, there remains the problem of understanding the end product and *rationale* of the transformation. In order to improve understandability, the refactoring sequences generated should be relevant to a specified goal of removing a given design flaw, i.e., contain only steps required to achieve that goal.

A problem with automated transformation in general, and refactoring in particular, is the traceability and understandability of the resulting design. Heavy modifications will make it more difficult to relate to previous designs and developers familiar with these may struggle to understand the new structure. We believe that both problems can be mitigated by exploiting the local

nature of refactoring operations, which affect only a certain part of the design while leaving the context unchanged.

For traceability, causal dependencies provide an explanation of why certain steps are required to perform later steps, thus reducing the problem to understanding the benefits of the final steps in a sequence. Thus, a representation of the search space by sets of transformation steps equipped with relations of causality and conflict is used to construct sequences where each step, directly or indirectly, contributes to the desired goal.

This requires a targeted approach, aiming at the removal of design flaws. The idea is that refactoring sequences should be *relevant* to the removal of the flaw identified. At the same time there may be more than one way to achieve this goal, requiring ranking and selection of the best sequence. In contrast to other search-based approaches, in this case metrics are not the main drivers of refactoring, but provide guidance in selecting between alternatives leading to the desired result with different costs and side-effects. The detection of design flaws [83] is beyond the scope of this work.

An overview of our goal-oriented refactoring approach is given in Figure 4.2 by means of a UML activity diagram. The following actions differentiate this approach from the one in Figure 4.1.

Identification of Design Flaws: We identify the relevant design flaws with

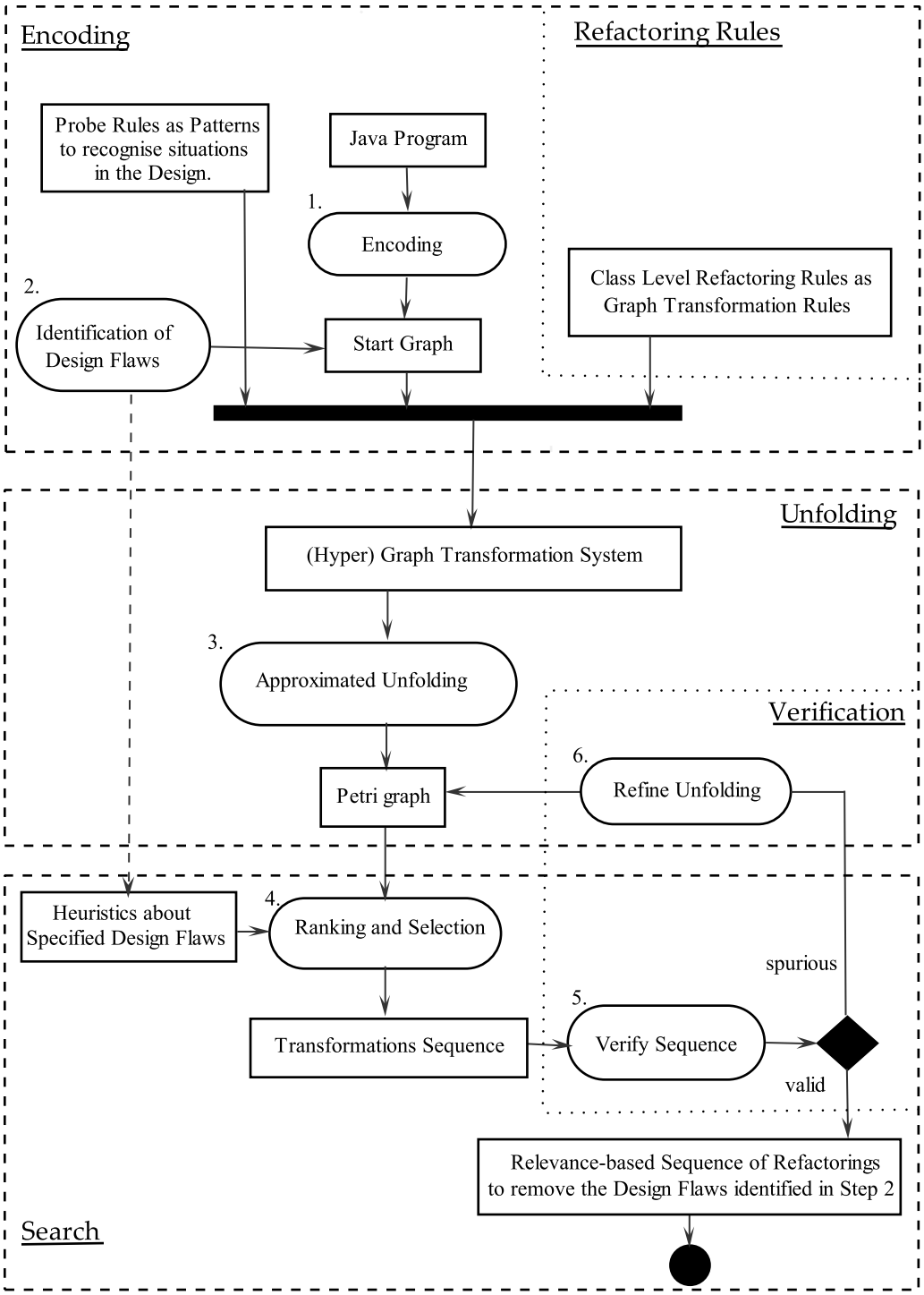


Figure 4.2: Abstract view of the Goal-oriented Refactoring Approach

the help of markers in the start graph encoding the Java program.

Search: Searching backwards from the identified design flaw, we find all sequences if refactorings removing this flaw and select one of them according to a specified metric.

Here the result is a sequence of transformations representing the best sequence of refactorings that is relevant to the desired goal of removing an identified design flaw.

4.2 Application to Running Example

The graphs we transform represent Java class structures, which can be visualised by class diagrams. As a running example we use the class diagram of a LAN simulation [61] in Figure 4.3. In addition to the standard notation we show call dependencies between methods (*calls this.send(p)*), variable access/update (*accesses Packet.sender* and *updates Packet.sender*).

A software developer could improve the structure of the design in Figure 4.3 by applying a variety of different refactorings. We consider the following set of refactoring operations [35].

- Extract Superclass, creating a common superclass for two existing classes, usually in order to encapsulate shared features.

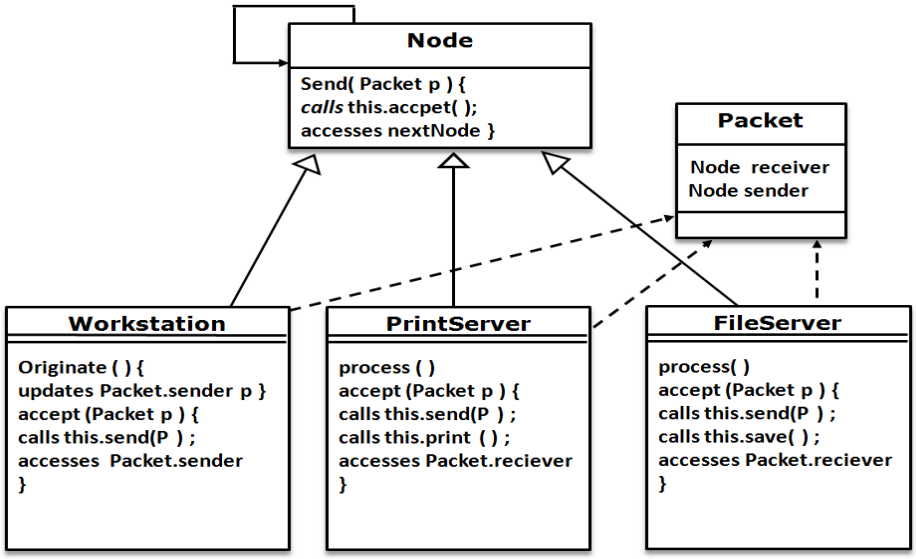


Figure 4.3: Simplified Class Diagram of LAN Simulation

- Add Parameter, introducing a new parameter for a method to make data access explicit.
- Pull Up Method, transferring a method from a sub to a superclass.
- Move Method, transferring a method to any other class.
- Encapsulate Attribute, to increase modularity by changing the visibility of an attribute in a class from public to private.

We formalise the local nature of individual refactoring operations by representing them as graph transformation rules [61]. Graphs representing class-level architectures of OO software are manipulated by rule-based transformations. Consider, e.g., the refactoring Extract Superclass describing the

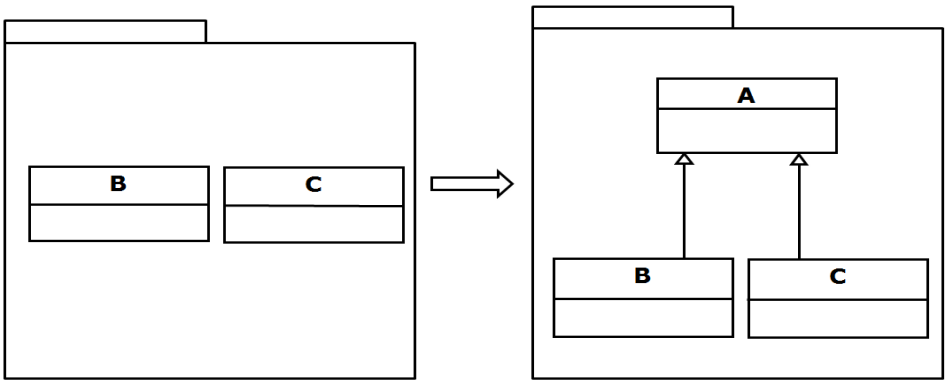


Figure 4.4: Rule for refactoring Extract Superclass, in class diagram notation

creation of a new superclass for two existing classes for the purpose of extracting common functionality. This operation is specified by the rule in the Figure 4.4, visualised in class diagram notation.

Rules can be applied in different orders and locations, giving rise to a number of refactoring sequences. Below we describe and motivate some of the steps for future reference.

t_1**:** Extract Superclass *Server* from classes *PrintServer* and *FileServer*. This will allow us to extract common methods into the new superclass.

t_2**:** Pull Up Method *accept* from classes *PrintServer* and *FileServer* into superclass *Server* created by t_3.

t_3**:** Move Method *accept* from class *PrintServer* to *Packet*, because it is more tightly coupled to that class (accessing its attribute).

t_4: Move Method *accept* from class $FileServer$ to class $Packet$, with the same motivation.

t_5: Encapsulate Variable *receiver* in class $Packet$, making the attribute private and creating setter and getter methods.

t_6: Add Parameter p of type class $Packet$ to method *process* in class $PrintServer$ to make explicit the access to instances of $Packet$.

t_7: Add Parameter p of type class $Packet$ to method *process* in class $FileServer$, for the same motivation.

Note that the transformations listed are not part of a single sequence. For example t_2, t_3 are in conflict.

Basically, in our encoding the identity of graph elements i.e., class, method and attribute etc., are represented by the nodes in the Petri graph. So, we consider every hyperedge (a first tentacles) attached to a node representing the identity of a node. Therefore, the pre- and post-sets for each transition in Petri graph are given by these hyperedges to workout the difference and intersection between these sets.

By analysing the resulting unfolding structure (Section 5.2) we can derive two relations on transformation steps, called causality and conflict [9]. A causal dependency $t_1 < t_2$ exists if t_2 cannot be executed before t_1 because

it relies on resources created by the first step. A conflict $t_1 \# t_2$ arises if both steps are enabled in the same graph G and require the same resources for their execution. That means, they prevent each other from occurring in the same sequence, but can occur individually.

Based on these two relations, reachable designs are represented by subsets of transformations $T \subseteq \mathcal{T}$ that are conflict-free and closed under causal dependencies, i.e., for all $t, t' \in \mathcal{T}$, T represents a state if:

- $t' \in T$ and $t < t' \Rightarrow t \in T$
- $t, t' \in T \Rightarrow \neg(t' \# t)$

The transformations enabled in T are those whose dependencies are satisfied by transformations in $\mathcal{T}$ and that are not in conflict with any transformation in that set.

That means, the empty set of transformations $\emptyset$ represents the initial design with no refactorings applied yet, while $\{t_1\}$ represents the design shown in Figure 4.5 obtained by applying Extract Superclass and, as we shall see later, $\{t_1, t_2\}$ represents the design with the duplication removed because it is closed under dependencies, whereas $\{t_3, t_4\}$ have a conflict.

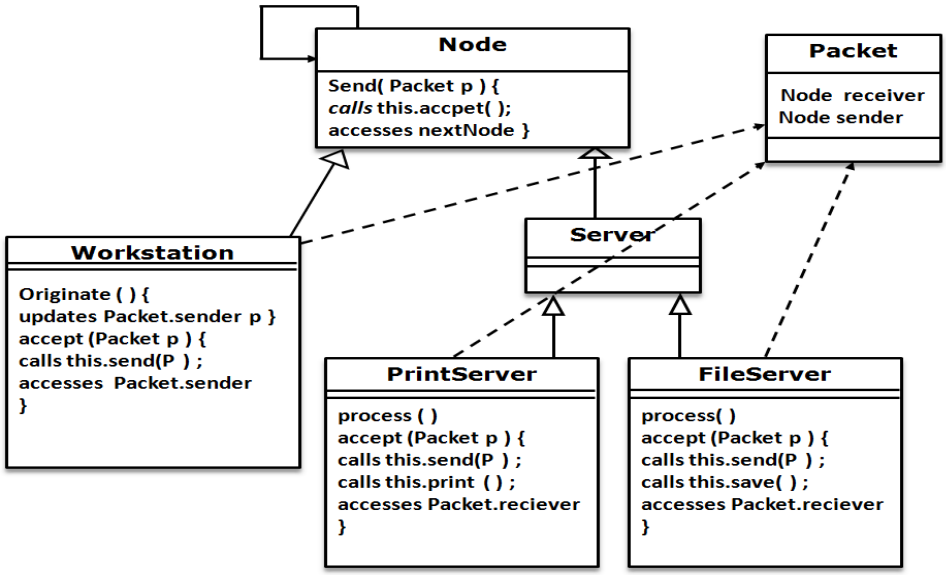


Figure 4.5: LAN Simulation after Extract Superclass

4.2.1 Search-based Refactoring

As outlined earlier, we use an implicit representation of the search space based on causality and conflict relations over rule applications representing refactoring steps such as t_1 to t_7 before. These partial orders are derived in two steps. First, the approximated unfolding of the grammar given by the start graph representing the initial design and the generic refactoring rules is produced and second, partial orders are derived by analysing the overlaps of the pre- and postconditions of these rules in the resulting Petri graph (Section 5.2). The hypergraph representation for the initial class model and refactoring Extract Superclass rule are depicted in the Figures 4.7 and 4.6 respectively.

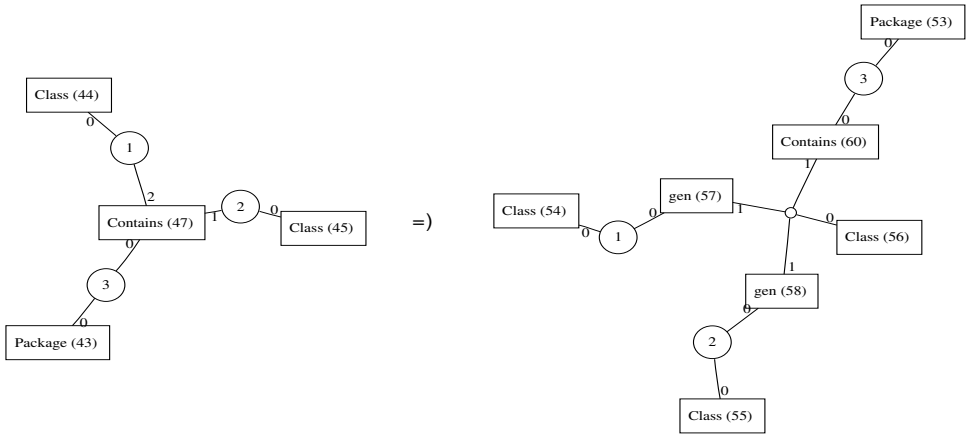


Figure 4.6: Hypergraphs for Extract Superclass Refactoring Rule

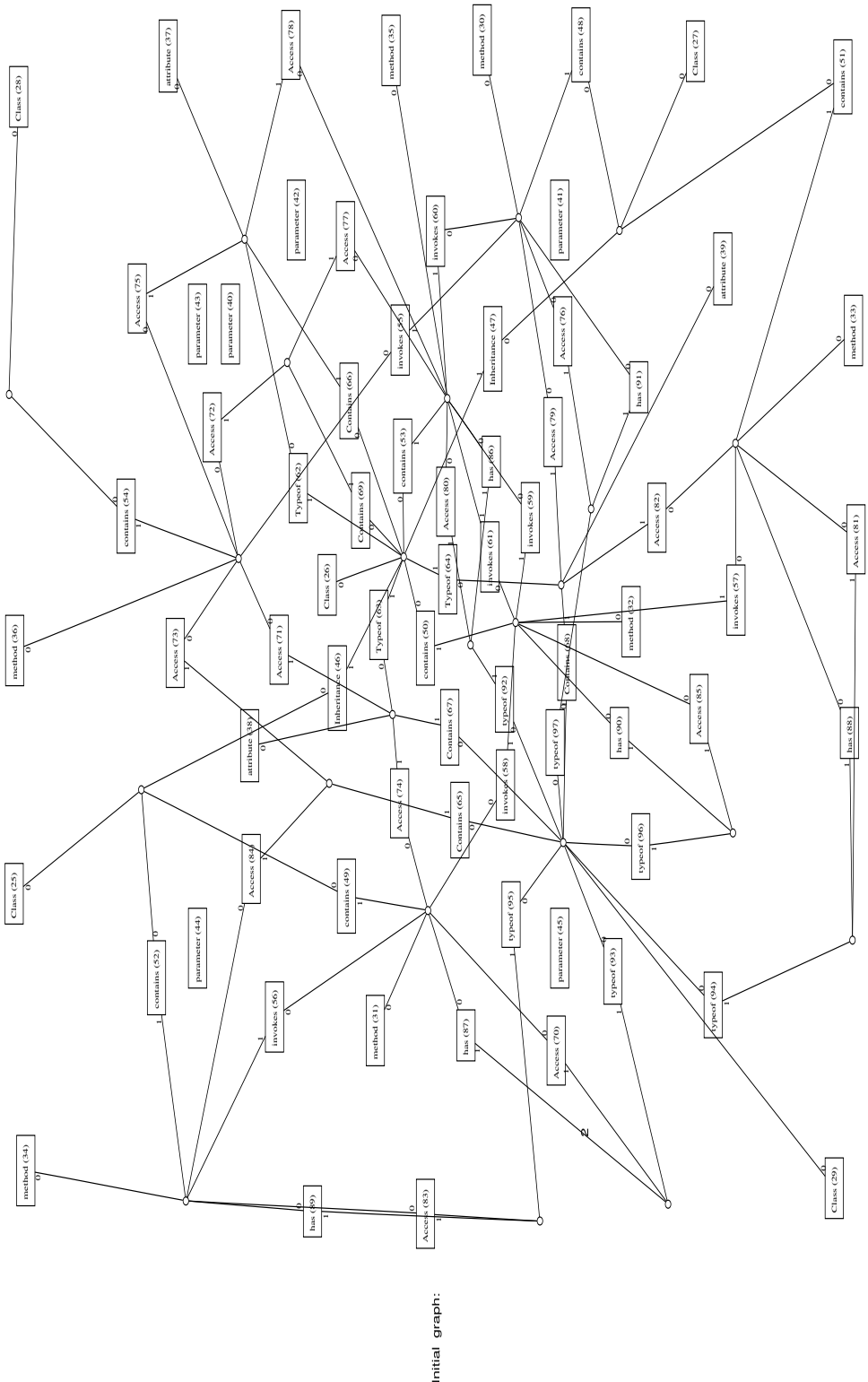


Figure 4.7: Initial Hypergraph

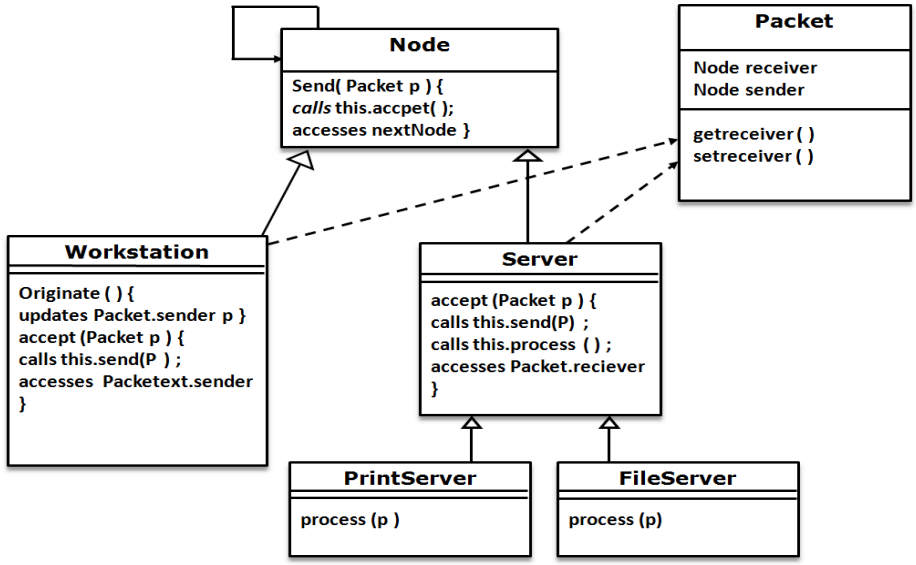


Figure 4.8: Final Class level diagram

Unfolding analyses a hypergraph grammar, starting with the initial hypergraph and producing a branching structure by applying all possible rules on the system. The resulting Petri graph presents the behaviour in terms of an over-approximation of its transformations and dependencies [6].

The Petri graph serves as input to a search problem. The optimisation yields paths representing sequences of transformations that are optimal to take the given design to a design of high(er) quality.

We make use of Ant Colony Optimisation (ACO) [28] metaheuristic search to find such a solution. The search space is defined by the associated set of all transformations, such as $\{t_1, \cdots, t_7\}$. Running our algorithm with five ants, each computing its own candidate solution, we obtain $\{t_5; t_7; t_6; t_3\}$,

$\{t_6;t_7;t_5;t_4\}$, $\{t_7;t_6;t_1;t_2;t_5\}$, $\{t_5;t_6;t_7;t_3\}$ and $\{t_5;t_6;t_7;t_1;t_2\}$.

We employ probe rules to represent patterns in order to evaluate candidate solutions. These probe rules will be used to define the objective function. The idea is to specify positive and negative patterns, so that their number of occurrences is maximised and minimised, respectively. As per the evaluation function the best path computed by the algorithm is $\{t_7;t_6;t_1;t_2;t_5\}$, representing an optimal sequence of refactorings. Since the unfolding represents an over-approximation, the existence of this sequences needs to be verified in the real model, possibly leading to a refinement of the approximation [51]. But our final optimal sequence exists in the real model and hence there is no need for refinement. The resulting class model after applying the optimal sequence of transformations is visualised in Figure 4.8.

4.2.2 Goal-oriented Refactoring

We will consider the same running example in order to illustrate the goal-oriented refactoring approach. Similarly, in this approach the Petri graph also serves as input to our search problem, but before constructing the approximated unfolding of the graph transformation system we identify the design flaws in the start graph representing the initial design. For example, we point out design flaws by markers such as *dup* and *pub* in the class diagram

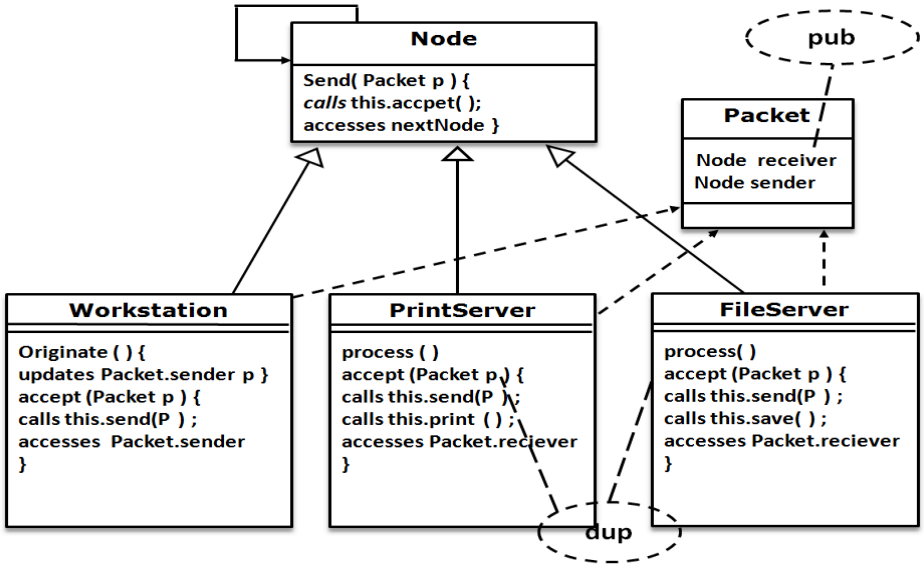


Figure 4.9: Simplified Class Diagram of LAN Simulation with markers.

of the LAN simulation in Figure 4.9, where *dup* refers to the duplication of a method and *pub* to publicly accessible variables.

In order to remove these flaws, we have to find a refactoring sequence based on a catalogue of basic refactoring operations [35], in our case consisting of Extract Superclass, Pull Up Method, Move Method, Encapsulate Variable and Add Parameter. They are specified by six graph transformation rules, which can be applied in different orders and at different matches, giving rise to transformations t_1 to t_7 described in the previous section. For example the application of t_1: Extract Superclass to the initial design has been shown in Figure 4.5.

The result of the unfolding of the grammar consisting of (a hypergraph

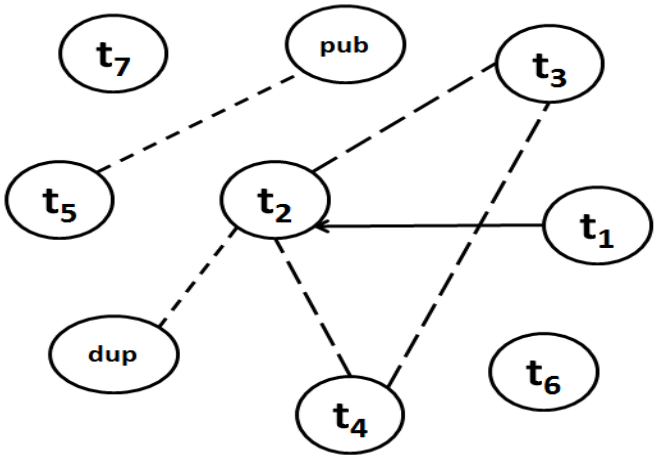


Figure 4.10: Graph of refactoring steps with their causal dependencies and conflicts

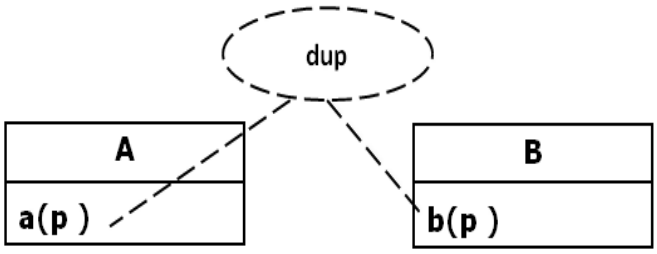
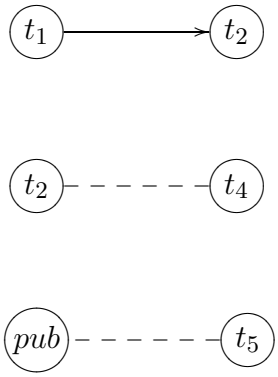


Figure 4.11: Pattern pointing out duplicated methods

representations of) our six refactoring rules and the initial LAN simulation design leads to a partial order structure as shown in Figure 4.10, where nodes represent refactoring steps and edges depict the following different relationships:



In addition, the dependency graph contains the occurrences of patterns *dup* and *pub* identifying designated design flaws. Such a pattern is expressed by a graph as in Figure 4.11 for duplication, whose purpose it is to define the location of the flaw by means of a mapping from the pattern graph into the design. It is important to remember that the detection of design flaws is not a concern of this work. We rely on external tools and the developer to tell us which flaws are to be removed. Formally a pattern graph P is regarded as a rule $P \to P$, which will be matched in the graph, but whose application does not have any effect. A conflict involving a pattern, such as that between *dup* and t_2, means that transformation t_2 removes the occurrence of the pattern, and thus the design flaw.

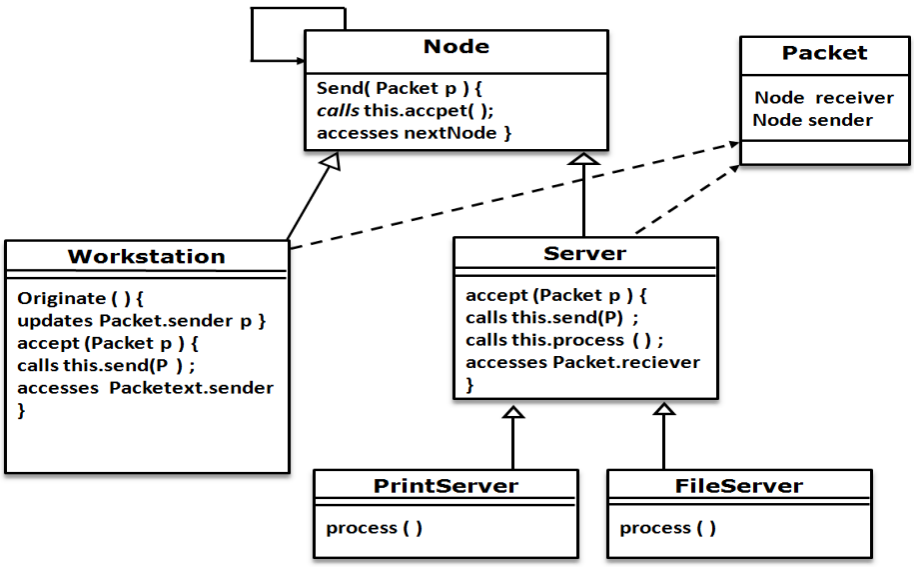


Figure 4.12: LAN simulation design after refactoring

The relational structure described in Figure 4.10 is used to find a refactoring sequence which removes one of the flaws identified while optimising the quality metrics used to guide the refactoring. The feasible solutions are constructed starting from the design flaw and selecting a conflicting transformation, which will be the final step in the sequence removing the flaw. The sequence is completed by computing the closure under causal dependencies. We start from the set of smells such that (*dup*, *pub*) representing the occurrences of the patterns identifying the designated design flaws with an empty solution.For each smell, we check the set of smell-killers (refactoring steps) in the dependency graph (Figure 4.10) in order to find a conflicting transformation, such as t_2 or t_5.

In our example, the closure of t_2 (the step removing the flaw labelled *dup* in the graph of Figure 4.10) under dependencies yields the set $\{t_1, t_2\}$. Any ordering compatible with the dependency relations yields a valid refactoring sequence. Figure 4.12 shows the resulting design after applying refactorings $t_1; t_2$ to the model in Figure 4.9.

Next, the feasibility of the sequence has to be verified on the actual design, because the unfolding used to construct the dependency structure produces an over approximation of the actual behaviour. If the sequence is rejected as spurious, the procedure is repeated based on a more detailed version of the unfolding, following the principle of a Counter-example Guided Abstraction Refinement (CEGAR) [51]. Here, in this example the final refactoring sequence is not spurious because it exists in the real model and therefore refinement is not needed.

Chapter 5

Analysis of Dependencies and Conflicts

In the following sections we study in more detail the technical realisation of the approach. As illustrated in Chapter 4 we use an implicit representation of the search space based on causality and conflict relations over rule applications representing refactoring operations. We derived these partial orders using the approximated unfolding of the grammar given by the start graph representing the initial design and the generic refactoring rules and by analysing the overlaps of pre and postconditions of these rules in the resulting Petrigraph.

5.1 Unfolding of the Refactoring Grammar

The approximated unfolding and its implementation in AUGUR2 [8] are defined for hypergraph grammars. Thus, we have to encode the initial design and refactoring rules into a hypergraph representations. As mentioned in Section 2.1 a hypergraph G is a tuple (V_G, E_G, C_G, l_G) where V_G and E_G are finite sets of nodes and edges respectively, $C_G : E_G \to V_G^*$ is a connection function, while l_G is a labelling function for edges [6]. The difference with the more common notions of typed or labelled graphs with binary edges is that, in hypergraphs, only edges are labelled, and each edge can be connected to a finite sequence of nodes, rather than just one source and one target. For example the hypergraph of the initial class model (Figure 4.3) is depicted in Figure 4.7. The idea is to introduce a node for each node in the original graph plus one edge to carry a label representing the type of the node. Additional binary hyperedges are introduced to represent edges in the original binary graphs. Rules undergo a similar transformation, but an additional restriction (imposed by the theory of unfolding) is that rules can delete and produce, but not preserve edges, while nodes cannot be deleted. The left-hand side of a rule must be connected [6]. This does not directly impact on the expressivity of the rules, but requires us to delete and regenerate edges that are meant to be preserved. The result for Extract Superclass is shown in the Figure 4.6.

Nodes of the left-hand side are mapped to those in the right-hand side with the same number, while the unnumbered nodes in the right are newly created. Edges in the left- and right-hand side are disjoint.

Given the hypergraph grammar, the unfolding starts with the initial hypergraph and produces a branching structure by applying all possible rules on the system at all possible matches. The resulting Petri graph contains both the graph structure of the system (essentially the union of all reachable graphs) and a Petri net with hyperedges as places and rule applications as transitions. The *approximated* unfolding creates a more abstract structure, potentially folding into one several graph elements or rule applications [6]. The result is an over approximation of the behaviour, i.e., spurious sequences may appear that do not exist in the actual behaviour. We use the AUGUR2 implementation of this construction [53] where initial hypergraph and rules are presented in the Graph Exchange Language GXL [79]. The output Petri graph produced is in GXL format as well [29].

5.2 Analysis of the Unfolding

From the pre- and post-conditions in this high-level representation we can extract causality and conflict relations on transitions. Using Peri net-like notation, we represent the pre- and post-sets for a transition t by ${}^\bullet t$ and $t^\bullet$,

respectively [8]. Then, in the obvious way, two transitions are in conflict, $t_1 \# t_2$, if and only if ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \phi$. They are causally dependent, $t_1 < t_2$, if and only if $t_1^\bullet \cap {}^\bullet t_2 \neq \phi$. The set of transitions $t_1, t_2, \ldots$ representing refactoring steps and relations $\#$ and $<$ provide the input to our search for an optimal sequence of refactorings.

Due to the fact that rules can't preserve hyperedges, if a hyperedge with the same label and connected to the same nodes occurs on both sides of a rule, it means that it has been deleted and regenerated in order to model an edge which should have been preserved.

Technically, a Petri graph has hyperedges as places and pre- and post-sets for a transition are given by these hyperedges. To avoid artificial conflicts and dependencies due to the encoding of edge preservation, we define more complex conditions. First, we define pre- and post-sets based on the nodes the hyperedges are attached to, making use of the fact that, in our encoding, each hyperedge considers the first attachment as an anchor. Then we perform the difference and intersection on these sets of anchor nodes.

$$ {}^*t = \{ fst(src(e)) \mid e \in {}^\bullet t \} \text{ and } t^* = \{ fst(src(e)) \mid e \in t^\bullet \} $$

where $src(e)$ is the list of nodes e is attached to and $fst(src(e))$ is the first element of this list.

We define sets of deleted, created and preserved nodes as follows.

1. $t^- = {}^*t \setminus t^*$, represents the nodes which are deleted.

2. $t^+ = t^* \setminus {}^*t$, represents the newly created nodes.

3. $\overline{t} = {}^*t \cap {}^*t$, represents the preserved nodes.

Causality and conflict relations are defined by:

- $t_1^+ \cap^* t_2 \neq \phi$ implies $t_1 < t_2$, that is, if an item is generated by t_1 and consumed by t_2, then t_1 must precede t_2 in any computation.

- $t_1^- \cap t_2^- \neq \phi$ and $t_1 \neq t_2$ imply $t_1 \# t_2$, that is, t_1 is in conflict with t_2.

These relations are the basis for an efficient representation of the search space in the next chapter.

For example, consider the set of refactoring steps mentioned in Section 4.2:

```
List of Transitions/Rules
=================================
_795247 | Add Parameter
_795246 | Add Parameter
_795245 | Move Method2
_795244 | Encapsulate Attribute
_795243 | Move Method1
_795242 | Pullup Method
_795241 | Extract Superclass
=================================
```

For the sake of brevity, we only consider the presets and postsets for transitions _795241, _795242 based on the anchor nodes to perform intersection and difference operations on them. Then, we compute the folowing added, delete and preserved sets for each transition allowing us to find the causal relation between them.

```
Presets and Postsets for Transition/Rules
==========================================
List of Presets for _795242:
 yn1_795129, yn2_795129, S0_795129, m1_795130, m2_795130
List of Postsets for_795242:
 yn1_795129, yn2_795129, S0_795129, m3_795130
------------------------------------------
List of Presets for _795241:
 wn1_795129, wns1_795146, wn2_795129, wns2_795146,
 m1_895130, m2_895130
List of Postsets for_795240:
 wn1_795129, wn2_795129, S0_795129, m1_795130, m2_795130
==========================================

Info - Added, Deleted and Preserved Sets
==========================================
Rule Id - _795242
Added  [m3_795130]
Deleted  [m1_795130, m2_795130]
Preserved  [yn1_795129, yn2_795129, S0_795129]
------------------------------------------
Rule Id - _795241
Added  [S0_795129]
Deleted  [wns1_795146, wns2_795146]
Preserved  [wn1_795129, wn2_795129, m1_795130, m2_795130]
------------------------------------------
```

Hence, the intersection between the added and preset of transitions _795241 and _795242, respectively, is not empty, which shows the causality between these two transitions.

Chapter 6

Refactoring as an Optimisation Problem

The performance of optimisation problems depends on the definition of the objective function [69]. The success of the optimization procedure relies on the selection of the objective function and its functional relationship to the control parameters [44]. Prior to explaining our optimisation problem, we describe the evaluation functions used in both approaches.

6.1 Evaluation Function

In order to formalise a notion of quality, we define probe rules as patterns to recognise situations that are desirable or to be avoided in object-oriented

designs. Then, we look for a solution having a maximum number of desirable and a minimum number of undesirable occurrences. Using the unfolding as underlying data structure, information about probe rule occurrences is available at little extra cost.

The question remains, how to find the occurrences of probe rules representing our pattern within the current graph. For example if G is a graph, the occurrence of a good pattern is represented by a match m for the identity rule that defines pattern p^+. A graph with 3 occurrences of p^+ is better than a graph with 2 occurrences. In Figure 6.1 and 6.2 we represent the probe rules in order to recognise the patterns that are bad or good respectively. The bad probe rules in Figure 6.1 will find all those classes with methods accessing attributes of a type of another class, whereas the good probe rules in Figure 6.2 will discover those classes with methods accessing attribute of a type of its own class.

For every probe p and solution s, we define $\#_p(s)$ as the number of occurrences of probe p in s. It will return negative integers for anti patterns. Assuming probes $p_1, p_2, \cdots, p_n$ (both positive and negative) the evaluation function is defined by $f(s) = \langle \#_{p_1}(s), \cdots, \#_{p_n}(s) \rangle$, returning a vector of probe counts.

Thus knowledge about good and bad patterns is embedded in the occurrence functions $\#_p$. We use pointwise extension of $\leq$ from integers to

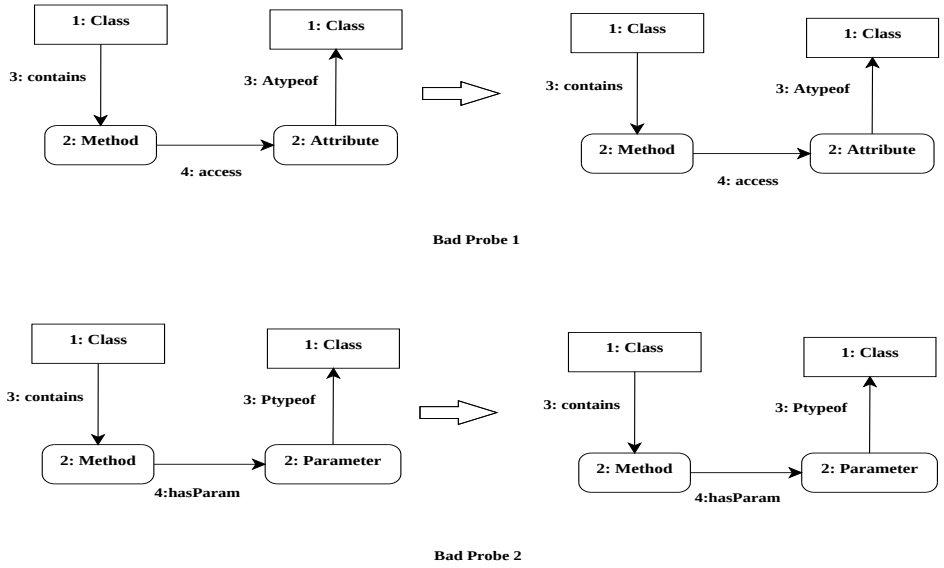


Figure 6.1: Probe rules representing bad patterns

vectors to define a partial order on the solutions, i.e., $v_1 \leq v_2$ if and only if $v_1[j] \leq v_2[j]$ for all $1 \leq j \leq n$. That means the relation must holds for every entry in the vectors to hold for the vectors in total.

The probability of choosing the next transformation depends on the quality of the successor solution, i.e., the number of occurrences of probe rules. Each ant will compute the probe vector while it moves from one solution to another in the search space.

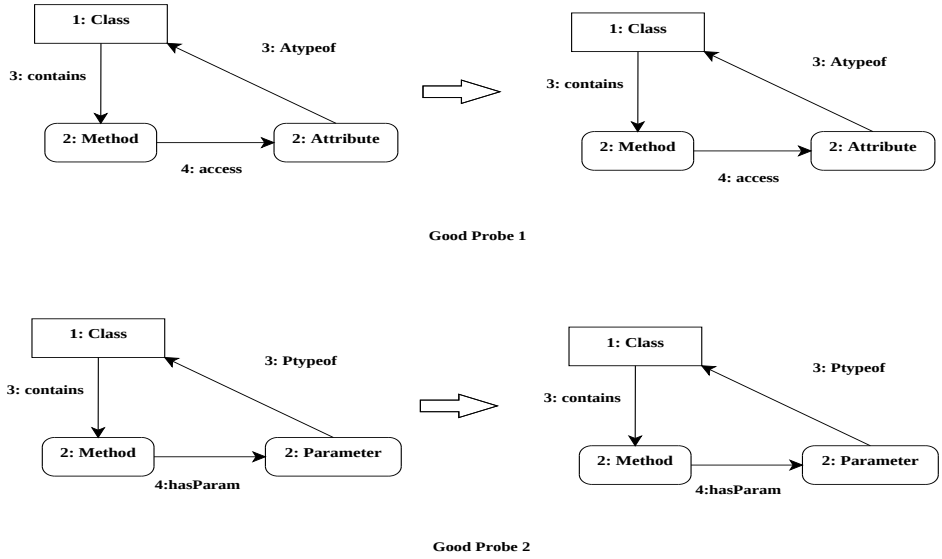


Figure 6.2: Probe rules represents good patterns

6.2 Search-based Refactoring as ACO Problem

We use the Ant Colony Optimisation (ACO) [28] metaheuristic to find an optimal refactoring sequence. The formal representation of an ACO problem has shown in Section 2.6

Artificial ants build solutions by moving randomly on the graph of components. Pheromone deposits are changed while ants move from node to node. Whenever the ants end their trip, we compute the best path according to the evaluation function. Then pheromone deposits are updated for all edges on that path [26].

We characterize search-based refactorings as an ACO problem using the formulation above.

1. The set of components C is given by the set of transformations $\mathcal{T}$. The edges E are the causality and conflict relations obtained from the unfolding construction. The conflict relations # requires mutual exclusion of refactoring steps and the causal relation $<$ requires two steps to occur in a certain order. Pheromone values τ_{ij} and heuristic values η_{ij} are associated with the edges of the graph.

2. The set of candidate solutions S consists of all refactoring sequences. The subset $\bar{S}$ of feasible candidate solutions is made up of all sequences that take the original design to a design of higher quality.

3. The set of optimal solution solutions S^* consists of the best refactoring sequences among the feasible solutions, i.e., those where the evaluation vector is maximal with respect to the partial order.

4. The evaluation function is defined by probe rules to recognise situations that are desirable or to be avoided in a design. Hence, probes are either good or bad in order to represent desirable or undesirable situations respectively (see Section 6.1).

Therefore, we consider a graph defined by the set of transformation steps

as nodes with edges representing relations derived from dependencies and conflicts as obtained from the unfolding construction. The problem is thus expressed as the search for an optimal sequence of refactoring steps applicable to the original system. The optimisation depends on an evaluation of sequences of refactorings representing candidate solutions, which takes into account both the cost of the refactoring transformations and the quality of the end result.

Intuitively, the search proceeds as follows. Locate ants on nodes representing initially independent transformations. At first, each ant will start with empty candidate solution ($|s| = 0$) and select any applicable refactoring from the available neighbours in the search space, where neighbours are enabled transformations which don't initiate conflicts and unresolved dependencies. Then the ant will look for the next possible move to construct a candidate solution and check its dependency with previous solution components in the sequence under construction. In this way, we can construct S as the set of all possible candidate solutions (sequences of nodes) until the termination condition (neighbours = $\emptyset$) is satisfied, i.e., when all ants complete their tours. After computing all feasible sequences from the search space, the optimal sequence will be chosen based on the evaluation function, balancing cost against quality as measured by the metrics.

We use a so-called Hybrid Ant System [36] where ACO is extended by

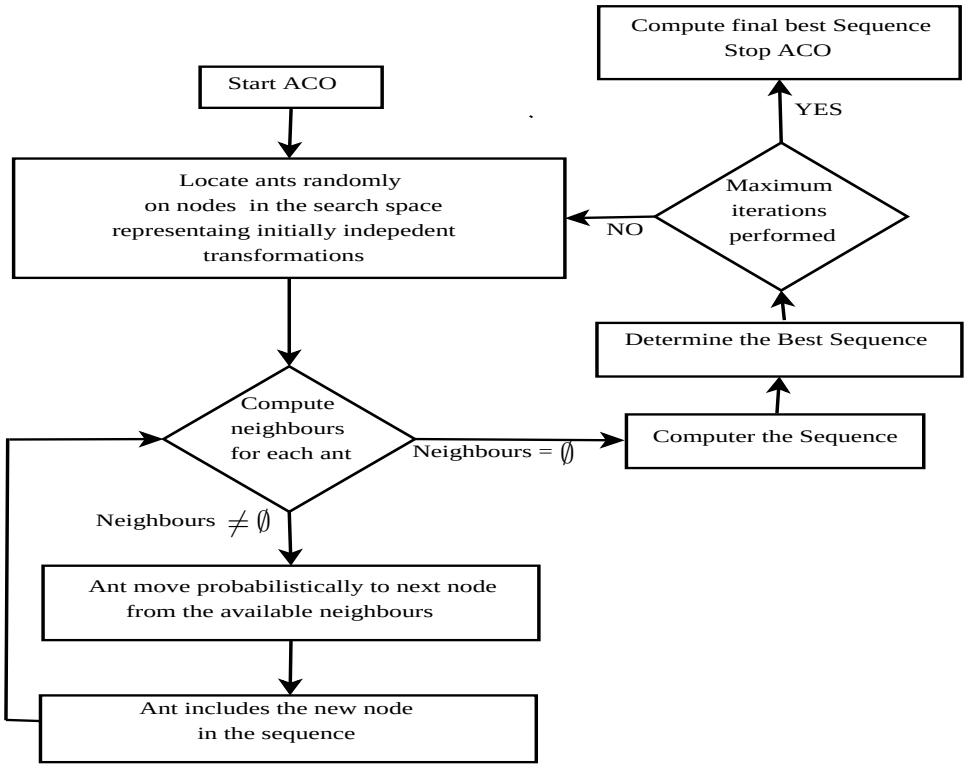


Figure 6.3: ACO-based algorithmic framework

local search. In particular, the Java framework by Chiricom [18] implementing [26] is used in order to solve a variety of ACS problems. We adapted this framework to an implicit representation of solutions based on our partial order model, deriving solutions and their local neighbourhood on the fly. The ACO based algorithmic framework for our problem has been depicted in the Figure 6.3.

Deriving Solutions and Transformations

Solutions S are subsets of transformations that are conflict-free and closed under causal dependencies as hown in Section 4.2.

The neighbourhood for a solution s is characterised by all transformations enabled in s. A transformation t is enabled in s if all its dependencies are satisfied by transformations in s and it is not in conflict with any transformation in that set. Adding such a transformation leads to a new solution $s \cup \{t\}$. While computing the neighbourhood for a solution in the search space, we need to check that the new transformation is not yet present in the solution. The conditions for enabled transformations ensure that the new solution is well-defined, i.e., the added transformation does not introduce conflicts or unresolved dependencies.

With these prerequisites the algorithms proceeds as follows.

- **Step 1:** We initialise each ant by assigning an empty state $s_0 = \emptyset$.
- **Step 2:** For each solution s, an ant will determine its local neighbourhood N_b by computing all enabled transformations in s, with successor solution $s_i = s \cup \{t_i\}$.
- **Step 3:** Ant moves to a neighbouring solution based on the pheromone values associated with the transformation.

- **Step 4:** After moving to a new solution, the ant will update the pheromone deposit locally.

- **Step 5:** The ant will stop when there are no more new transformations to be added, i.e., $N_b = \emptyset$.

  At the end a global update will take place to increase the pheromone deposits on all arcs leading to success, leaving the rest unchanged.

- **Step 6:** An evaluation function $f(s)$ computes the probe vector for each solution s.

Hence, the end result is the best path computed by the algorithm, representing an optimal sequence of transformations for a given initial design.

6.3 Goal-oriented Refactoring as a Search Problem

In this version of the problem our search is goal oriented in order to find the optimal sequence of transformation, addressing a specific design flaw. Like for search-based refactoring, we use the implicit representation of the search space by sets of transformation steps equipped with relations of causality and conflict. We exploit the relational structure obtained from the partial order analysis to find refactoring sequences that are relevant to the goal

of removing a given design flaw. We present goal-oriented refactorings as a search problem using the same construction graph as mentioned in the previous approach.

1. **Step 1:** We assign a design smell to each solution s_0 initialised by an empty set ($s_0 = \emptyset$).

2. **Step 2:** Compute the smell-killers, i.e., conflicting transformations for each smell.

3. **Step 3:** A feasible solution s is completed by computing the closure under causal dependencies. Hence, a finite set S is made up of all sequences that end with a transformation removing the identified flaw.

4. **Step 4:** An evaluation function $f(s)$ computes the probe vector for each solution s.

Therefore, the search space is associated with a graph defined by the set of transformation steps as nodes with edges representing relations derived from dependencies and conflicts as mentioned in the Figure 4.10. We will consider pattern rules representing design flaws in the given system along with the proposed set of refactorings as a set $\mathcal{T}$ (as shown in the Figure 4.9). The sequences are constructed starting from a design flaw taken from the set of design smells and selecting conflicting transformations for each smell in order

to compute the set of smell-killers. Each sequence is completed by computing the closure under causal dependencies. Such sequences are relevant to the removal of the flaw identified and there may be more than one way to achieve this goal, requiring ranking and selection of the best sequence.

Chapter 7

Verification of the Transformation Sequence

Since our approach is based on an over-approximation of graph transformations, it might lead to spurious solutions which have to be filtered out. Hence the feasibility of the final transformations sequence has to be verified on the actual design. If the sequence is rejected as spurious, the refinement procedure is repeated based on a more detailed version of the unfolding, following the principle of Counter-example Guided Abstraction Refinement (CEGAR) [51].

CEGAR has been used successfully for the purpose of verification [19]. The idea is to start with coarse initial over-approximation of a system and try to verify a certain property. If the property cannot be verified, there is a

run, i.e., sequence of transitions in the Petri graph that violates this property, known as counter-example [51]. We make use of CEGAR in order to realise the existence of the optimal sequence in the real model, possibly leading to a refinement of the approximation.

AUGUR2 has the functionality to specify prohibited paths in graphs using regular expressions with the set of hyperedge labels as the alphabet. It produces conditions on the marking of the hypergraph, which guarantee that a path in the hypergraph matches the regular expression. This regular expression illustrates forbidden paths which should not occur in any reachable graph [51].

It is a "known" limitation in AUGUR2 that complex regular expression won't work. The reason is that the over-approximation might not be caused by a rule application, but by a false positive for a subgraph satisfying the regular expression. For this case there are no refinement rules. If however the regular expression consists of only one edge, there can be no such false positive.

Spurious counter-examples always arise from node fusion in the over approximation. Edges are also fused, but simple edge fusion will not cause any spurious runs. However, if two nodes are merge that "should" not be merged, new left-hand sides might appear, more rules might be applicable, etc. If a spurious run has been found, the tool detects the cause of the spuri-

ous behaviour, i.e., the merging of two nodes. Then, in the next abstraction refinement loop the merged nodes are deliberately kept apart.

In our case this functionality is not directly accessible because there is no general facility in AUGUR2, where we provide a sequence of transformations and it checks for us whether the sequence is real or spurious. However, for smaller examples we are exploiting this functionality in AUGUR2 by producing a counter-example manually from the transformations sequence computed by the search and then replacing the *transpath* file with that sequence accordingly. Then we can check the validity of our run via the AUGUR2 functionality by pressing the "Unfold" button. A *transpath* is a file, which contains trace for a coverable marking in the Petri net.

The implementation of this functionality is sketchy and only serves to show the principle and will be of interest in future research for verification of sequences in the case of larger case studies.

Let us consider the final optimal sequence of Section 4.2.1 for the purpose of verification. We generate a counter-example by introducing a special rule $Error$ with a single 0-ary edge (labelled "$0'Error'0$" in the right-hand side) along with the refactoring rules mentioned in Section 4.2. This edge labelled "$0'Error'0$" of the approximation (Petri graph) can be covered by firing transition $Error$.

After constructing the approximated unfolding of the graph grammar, we

specify a regular expression "$0'Error'0$" in the property box of the tool in order to check the feasibility of the final transformation steps. This regular expression is converted into a marking of the Petri graph via scenario *property2marking*, , which is a special algorithm. The regular expression is saved in the file *work/regexpr* and the resulting marking is written to *work/marking*.

After this we check if this marking is coverable in the underlying Petri net. Towards this goal we call the scenario *cover* with a standard coverability algorithm, which checks whether a marking can be covered (as depicted in the Figure 7.1). If the marking is coverable in the Petri net, the trace is saved in the file *work/transpath*. Now, we replace the *transpath* with our own final sequence of transformations and try to check its validity in the actual model.

Hence, if the trace (sequence of transformations) is real it will appear in the window as "The counter-example is real" as shown in the Figure 7.2 otherwise it will show "The counter-example is spurious".

In case the trace is spurious, i.e., it has no counterpart in the original grammar, to eliminate the spurious run, the tool detects the cause and avoids it in the next iteration [51] to compute a refined Petri graph.

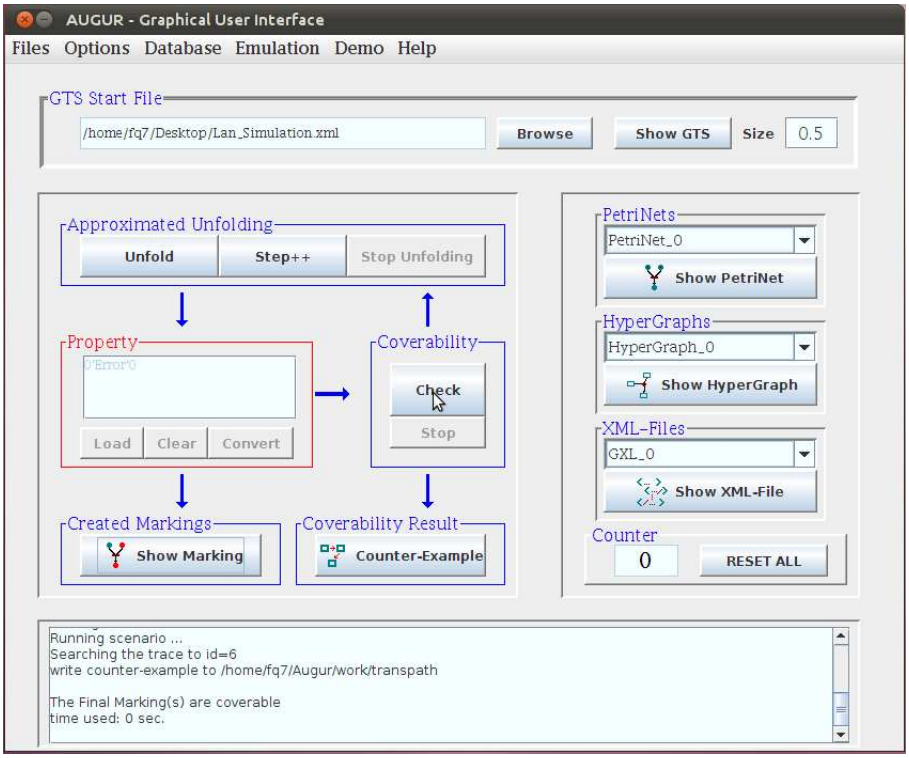


Figure 7.1: Screen-shot 1 of GUI Panel of AUGUR2

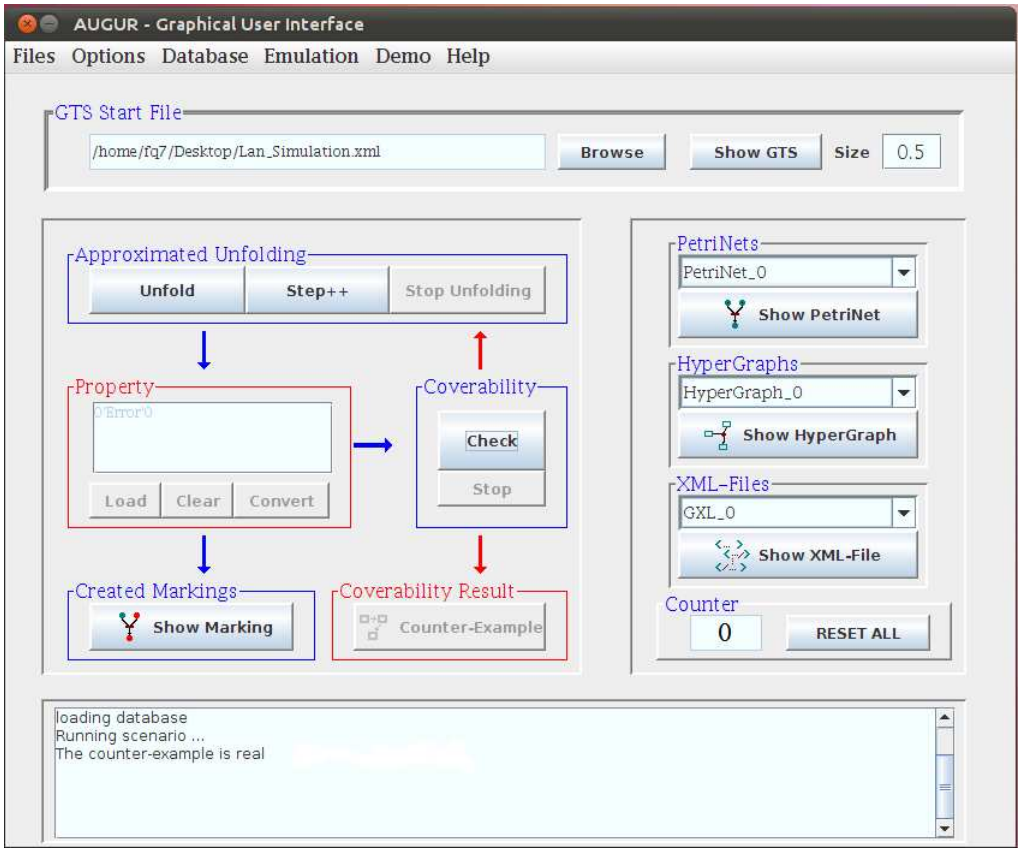


Figure 7.2: Screen-shot 2 of GUI Panel of AUGUR2

Chapter 8

Implementation

In order to use the unfolding technique, we encode the initial class model and graph transformation rules for refactoring steps into the GXL format required by AUGUR2. This is accomplished with the help of the environment called InFusion [47] and a subsequent transformation of the resulting MSE [56] file into GXL. The result represents the start graph of the hyper graph grammar to be unfolded. The rules of the grammar formalising the refactoring operations are derived from the standard catalogue [35] shared across all Java programs. AUGUR2 constructs the approximated unfolding of a system [8], producing a Petri graph to serve as input to the ACO-based search algorithm.

8.1 Translating Java Source Code into GXL

8.1.1 Java to MSE

We transform Java source code into MSE with the help of an integrated environment called inFusion [47], which can be used for in-depth code and architectural analysis of object-oriented and procedural software systems written in C, Java or C++.

We consider the same running example as in Chapter 4. The Java source code represents a simulation of a Local Area Network (LAN) that has been used at various universities to teach object-oriented design and refactoring [23]. For the sake of brevity, we will envisage the simplified version of the class hierarchy for this LAN in Figure 8.1 in order to see how the resulting MSE file corresponds to the actual model.

We infer information from the MSE file about the classes, methods, attributes and parameters along with data call and access dependencies etc. Figure 8.2 depicts the core entities and relations of our metamodel for extracting a GXL from a MSE file.

The following are excerpts of the MSE file generated from the Java source code. Each element and relation has its own id (integer) and name (string). Below we cite some of the representations for classes, methods, attributes and some of the access relations between different entities depicted in Figure 8.1.

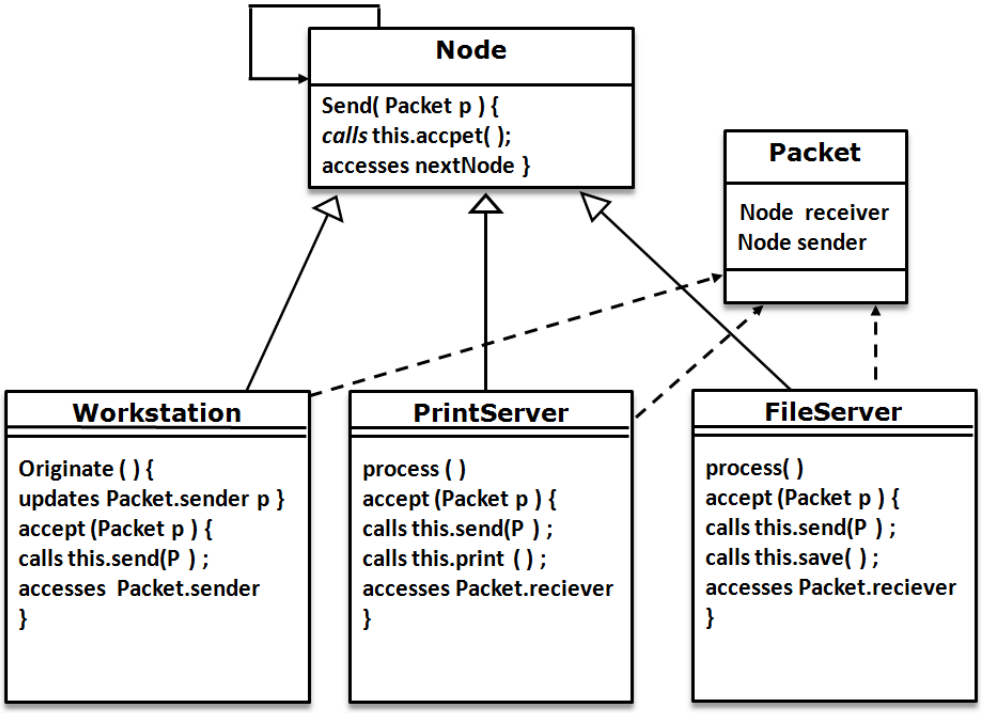


Figure 8.1: Simplified Class Diagram of LAN Simulation

1. The classes are represented with their names and ids. Below are the representations for the classes Node, PrintServer, Workstation.

```
(FAMIX.Class (id: 11)
(sourceAnchor (ref: 174))
(container (ref: 1))
(name 'Node')
....
)
(FAMIX.Class (id: 72)
(sourceAnchor (ref: 178))
(container (ref: 1))
(name 'PrintServer')
```

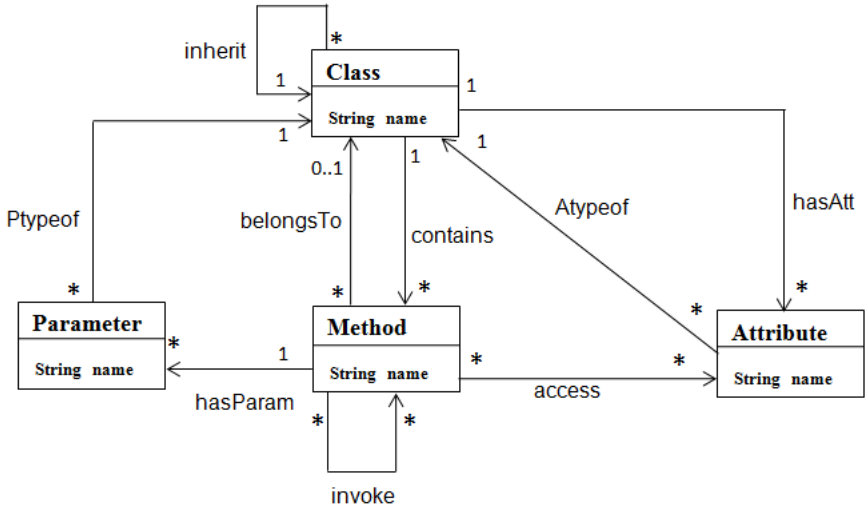


Figure 8.2: Metamodel for extracting information from MSE file

```
....
)
(FAMIX.Class (id: 95)
(sourceAnchor (ref: 182))
(container (ref: 1))
(name 'Workstation')
....
)
```

2. The inheritance relation has its own id and refers to individual ids for subclass and superclass, e.g., Class Node inherits Class PrintServer.

```
(FAMIX.Inheritance (id: 179)
(subclass (ref: 72))
(superclass (ref: 11))
....
)
```

3. Methods have their own ids and names, along with the information about their parents and declared types. Beside its own id, method invocations also contain ids for both methods, i.e., sender and candidate.

```
FAMIX.Method (id: 74)
(sourceAnchor (ref: 192))
(parentType (ref: 72))
(declaredType (ref: 15))
(name ‘print’)
....
)
(FAMIX.Method (id: 81)
(sourceAnchor (ref: 193))
(parentType (ref: 72))
(declaredType (ref: 15))
(name ‘accept’)
....
)
(FAMIX.Invocation (id: 86)
(sender (ref: 81))
(candidates (ref: 74))
....
)
```

4. Access relations among methods and attributes can be characterised in the same way, e.g., an attribute named ‘contents’ of type class (id:22) is accessed by method (id:74):

```
(FAMIX.Attribute (id: 41)
(name ‘contents’)
```

```
(parentType (ref: 22))
(declaredType (ref: 32))
....
)
(FAMIX.Access (id: 79)
(variable (ref: 41))
(accessor (ref: 74))
)
```

5. Method access parameter relations are represented in the same way, e.g., the parameter 'p' of type class (id:22) is accessed by method (id:74):

```
(FAMIX.Parameter (id: 76)
(name 'p')
(parentBehaviouralEntity (ref: 74))
(declaredType (ref: 22))
)
(FAMIX.Access (id: 80)
(variable (ref: 76))
(accessor (ref: 74))
)
...
```

8.1.2 MSE to GXL

The class structure of a given Java program (excluding method bodies, but keeping call and data access dependencies) is encoded into the GXL format required by the analysis tool (AUGUR2). Hence, the resulting GXL docu-

ment will represent the start graph of the hypergraph grammar containing all entities mentioned in the metamodel (as shown in Figure 8.2). There is a node for each node in the original graph along with one edge to carry a label representing the type of the node. Furthermore, binary hyperedges are introduced to represent edges in the ordinary binary graphs. The translation process is mostly automatic. The nodes of the object-based graph grammar are converted into hyperedges, whereas the connections between nodes become nodes in the hypergraph.

The transformation of a MSE file into the GXL format is done with the help of our MSE Reader designed in Java. It takes a MSE file as an input and produces the final output as an initial hypergraph represented in GXL. By the MSE Reader, the whole MSE file is considered as consisting of a string of tokens. It has specific identifiers for each element such as $FAMIX.Class$, $FAMIX.Method$ and $FAMIX.Inheritance$, etc., as depicted in the previous section. We search for these tokens to get the respective classes, methods, attributes and relations such as inheritance, invocation and access etc. We can utilize tokens to split a string into sub-strings.

To read the MSE file, we employ the $TextTransformer$ class in order to generate tokens out of that file. For this purpose, we use a class called $StreamTokenizer$ and pass reference of a $TextTransformer$ in its constructor. Along with the input string, we need to specify a string that contains

delimiters (characters). The $TextTransformer$ class scans the MSE file line by line to match keywords like $Class$, $Method$, $Inheritance$ etc. We create an object of the $BufferedReader$ class and pass the $InputStream$ object into it, which will read the MSE file at run time. Once a match is found, the attributes of the corresponding object are read and stored in maps. The process of searching attributes is the same as for searching the keywords such as $Class$, $Method$ etc., explained above. We make use of a DOM parser to prepare the XML document. The DOM parser is used here for the sake of simplicity in XML tree generation. The header of the XML document is a standard GTS element with attributes.

For each $KeyMap$, we create an XML element of type *node* and set the id as a value concatenated with "_". For example, when generating XML constructs for a class from MSE an file, we create elements such as relation, attribute, string and relend. In each of these elements are placed attributes like id, name and class, etc., depending on the type of element. These elements are then appended to one another to create a tree structure. So, every $Class$ element has child nodes *rel*, *attr*, *string* and *relend*. The maps relate the property values through a key-value relation. The key is the unique id against which a value is stored. $KeyMaps$ are made for all the properties required, for example, $attribute-Map$ denotes the $KeyMap$ for all attributes. Once the $KeyMaps$ are ready, we start populating an

XML document. The XML constructs are generated based on a formally agreed contract. The XML structure is prepared using element tags and the attribute values are set. The values are chosen from the appropriate $KeyMap$ and substituted as string values.

The relations are mapped to the classes/attributes. Along with this, we map properties as well, for example, attribute has a type and it also belongs to a class. We take care of these kinds of relation while scanning a MSE file. Individual $KeyMaps$ are used to store these relations. When an XML element for an access relation is generated, we check which attributes have this value of access id from the attribute $KeyMap$. Since we already have knowledge regarding the access elements and the child nodes it contains, we pull out the relevant information from the appropriate maps and fill in the XML elements, attribute values. Likewise, a similar procedure is used to generate other XML elements.

Below are the excerpts from the final output as an initial hypergraph represented in GXL syntax extracted from the Java source mentioned earlier in this section. The key word *rel* (= relation) defines a hyperedge, whereas *relend* introduces the tentacles of a hypergraph. Each entity of the class diagram will have its own node with unique id's:

- Each element like class, method and attribute, etc., has its own id.

```
<GTS id="LAN Simulation">
<Initial>
<Graph edgeids="true" edgemode="undirected"
hypergraph="true" id="Start Graph">
<node id="22_Packet"/>
<node id="9_Fileserver"/>
         ...
<node id="13_save"/>
         ...
```

- The hyperedge representing the identity of a node:

```
<rel id="n_22">
<attr name="label">
<string>Class</string>
</attr>
<relend id="n_relend1 "startorder="0" target="22_Packet"/>
</rel>
        ...
<rel id="method_13">
<attr name="label">
<string>method</string>
</attr>
<relend id="method_relend1" startorder="0" target="13_save"/>
</rel>
        ...
```

- The inheritance relation between two classes:

```
<rel id="R_173">
<attr name="label">
<string>Inheritance</string>
</attr>
```

```
<relend id="R_relend1" startorder="0" target="9_Fileserver"/>
<relend id="R_relend2" startorder="1" target="11_Node"/>
</rel>
```

- The invocation relation between two methods:

```
<rel id="invokes_67">
<attr name="label">
<string>invokes</string>
</attr>
<relend id="invokes_relend2" startorder="0" target="58_send"/>
<relend id="invokes_relend1" startorder="1" target="51_accept"/>
</rel>
```

- The access relation between method and parameter:

```
<rel id="access_68">
<attr name="label">
<string>Access</string>
</attr>
<relend id="access_relend2" startorder="1" target="62_p"/>
<relend id="access_relend1" startorder="0" target="58_send"/>
</rel>
```

8.2 Data Structure for Petri Graphs

A Petri graph [6] is a finite data structure which records the behaviour of a graph transformation system. It combines hypergraphs with Petri nets used to approximate the behaviour. The hyperedges of the graph component are at

the same time the places of the Petri net. The GXL representation produced by AUGUR2 [29] is in a low level format and problem independent. It only knows about graph elements and their attributes, but not about transformations. To create a problem-specific data structure to allow for dependency analysis, we have to extract information about rules and transformations, then we can derive conflict and dependency relations by comparing the pre- and post-sets of transformations.

We need a data structure specific to our problem, which is not entirely generic and can be optimized to the purpose at hand because we need an efficient algorithm to run on it. In the process of unfolding, we have transformations, hyperedges, nodes and associations which are called source and they are ordered. Indirectly, there exist relations such as causality, conflict and concurrency but they are not in the basic data structure. We can get this information from the pre-set and post-set conditions of a transformation. The class diagram in Figure 8.3 provides the conceptual data model for the unfolding process, in the sense that it is a set of transformations and each transformation has pre-conditions and postconditions, where pre-conditions and post-conditions consist of sets of hyperedges. A Java object graph representing an instance of this model is extracted from the GXL representation produced by AUGUR2.

The metamodel provides an object oriented abstraction, enabling us to

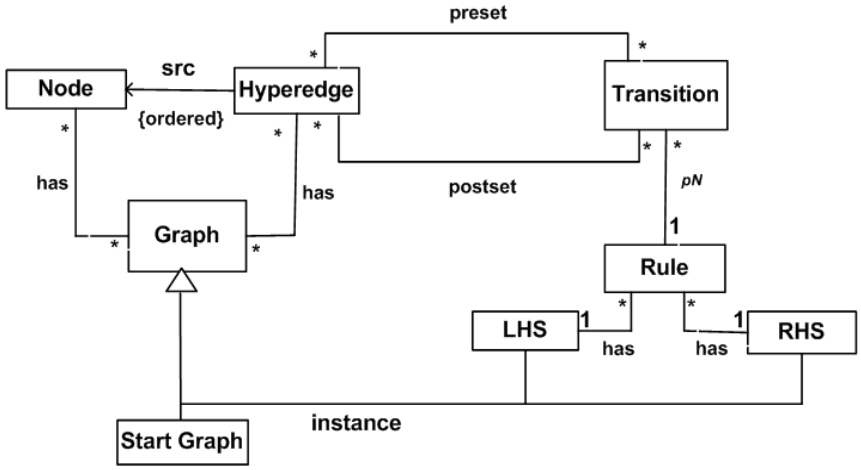


Figure 8.3: Metamodel for the Unfolding process

extract explicit information to run our optimisation algorithm. Hence, we can use the unfolding as an implementation for that data structure using efficient queries to extract concrete information about dependencies and interrelationships among the solutions in the search space of our algorithm. The UML component diagram shown in Figure 8.4 describes how our search algorithms will access the information from the output file (Petri graph) of an unfolding process.

Our GXL Reader enables us to extract explicit information about transformations and their dependencies from the approximation result (Petri graph). It computes a dependency table, consisting of causal dependencies and conflicts, derived directly from the Petri graph, and serves as an input to our search problem.

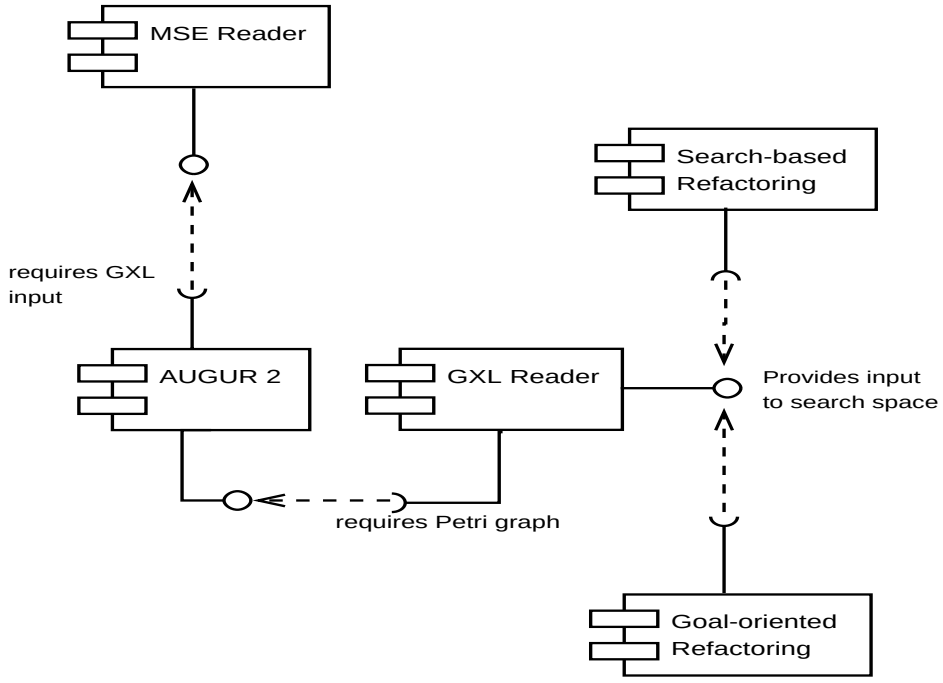


Figure 8.4: Relationships among the components in our methodology.

Initially, the GXL output file (Petri graph) is read through an XML DOM parser, which converts it into an XML DOM object, that can be accessed and manipulated. Hence, we can navigate the Petri graph representation in the memory at object level, in order to capture and move the required information through different queries into the higher level model, which are indirectly present in the output of an unfolding process. We need those nodes of the GXL file that have *transition* as their attribute name along with the interrelationships among them in order to compute a search space for our optimisation algorithm.

8.3 Search Techniques

As we mentioned in the Chapter 4 we are dealing with two different problems in search-based refactoring, i.e., automation and traceability. To address these problems, we implement two different search techniques. Below, we illustrate the implementation of both.

8.3.1 Search-based Refactoring as ACO problem

We use an object-oriented framework by Ugo Chirico written in Java to employ Ant Colony Systems [18]. After some modifications the proposed framework is reused and adapted to implement search-based refactoring. In Section 2.6 we have already discussed and presented the general framework of Ant Colony Systems but here we present the actual implementation of the algorithm. There are three main components which collaborate in an Ant Colony System: a set of ants, an ant colony and an ant graph [26].

Ant is an abstract class which employs the general behaviour of an artificial Ant. There is a concrete derived class called *AntforRef*, which implements abstract methods corresponding to the ACO rules, i.e., the *stateTransitionRule*, the *localUpdatingRule*, *compare* and *end*. Each ant works in its own thread launched by the *start* method. Java implementation for *Ant* class is based on the following algoritham.

```
Iterate
 Call State Transition Rule
 Update the sequence
 Call Local Updating Rule
Until End of Activity returns true
Call Comparison Function
If the current sequence is better than the best sequence
then update the best sequence.
```

AntColony is an abstract class employing the general behaviour of an Ant Colony System and it should be specialized to the specific problem as well. The abstract methods *createAnts* and *globalUpdatingRule* are implemented in the concrete derived class *AntColonyforRef.*

AntGraph is a class which understands the features of a graph specific for Ant Colony Systems.

8.3.2 Goal-oriented Refactoring as search problem

In this approach, we make use of the same construction graph like in the previous case but here our search is goal-oriented. There are two main entities, i.e., a set of smells and a corresponding set of smell-killers for each smell.

A high-level view of our search algorithm is as follows:

1. Initialisation

   for i=1 to n do

$s_{m_i} \leftarrow s_0$

end for.

Each solution is initialised by an empty set ($s_0 = \emptyset$).

2. The conflicting transformations, i.e., smell-killers are assigned for each design smell and the casual history of every smell-killer is computed.

```
Set<String> smells = smellKiller.keySet();
int count = 0;
for(String sm : smells)
{
   List<String> smellK = smellKiller.get(sm);
   for(String sk : smellK)
   {
 List<String> list = getParentList(sk);
 sequence.add(list);
    }
count++;
}
```

3. A candidate solution s is constructed starting from the design smell s_m. The solution is completed by computing the closure under causal dependencies.

4. An evaluation function $f(s)$ computes the probe vector for each solution s.

Chapter 9

Proof of Concepts

In this section, we consider the following Java program in order to evaluate our methodology. The main purpose of this evaluation is to find out whether our approach could find meaningful refactorings in the original system structure. Secondly, we are interested to see how much improvement is achieved after the application of the final optimal sequence of refactorings suggested by the optimisation.

9.1 Patient Information System (PIS)

We start with a simple example representing a small software system aimed to support a doctor's surgery. The system can register patients (these are either categorised as children, adult or elderly). It can print out prescription

forms for drugs, and compute the Body Mass Index (BMI) recommendations. The system is not very well designed; there are "bad smells" in the system. As we are interested in improving the class structure of a system, we consider the following set of generic refactorings from the catalogue [35], which could be helpful in aiding a software developer to improve the structure of the design in Figure 9.1. For the sake of simplicity readable, we mention only the initial design with the help of UML Class diagram, the call dependencies between methods and associations between classes are ignored in Figure 9.1.

- Extract Superclass, creating a common superclass for two existing classes, usually in order to encapsulate shared features.
- Pull Up Method, transferring a method from a sub to a superclass.
- Move Method, transferring a method to any other class.
- Encapsulate Attribute, to increase the modularity by changing a visibility of attribute in a class from public to private.

We provide the initial hypergraph representing the initial design/hypergraph rules modelling refactoring operations as shown in Section 2.3. Probe rules representing patterns as shown in Section 6.1 complete the input to AUGUR2 in order to construct the approximated unfolding [53]. This approximation will allow us to analyse the concurrent behaviour of the system

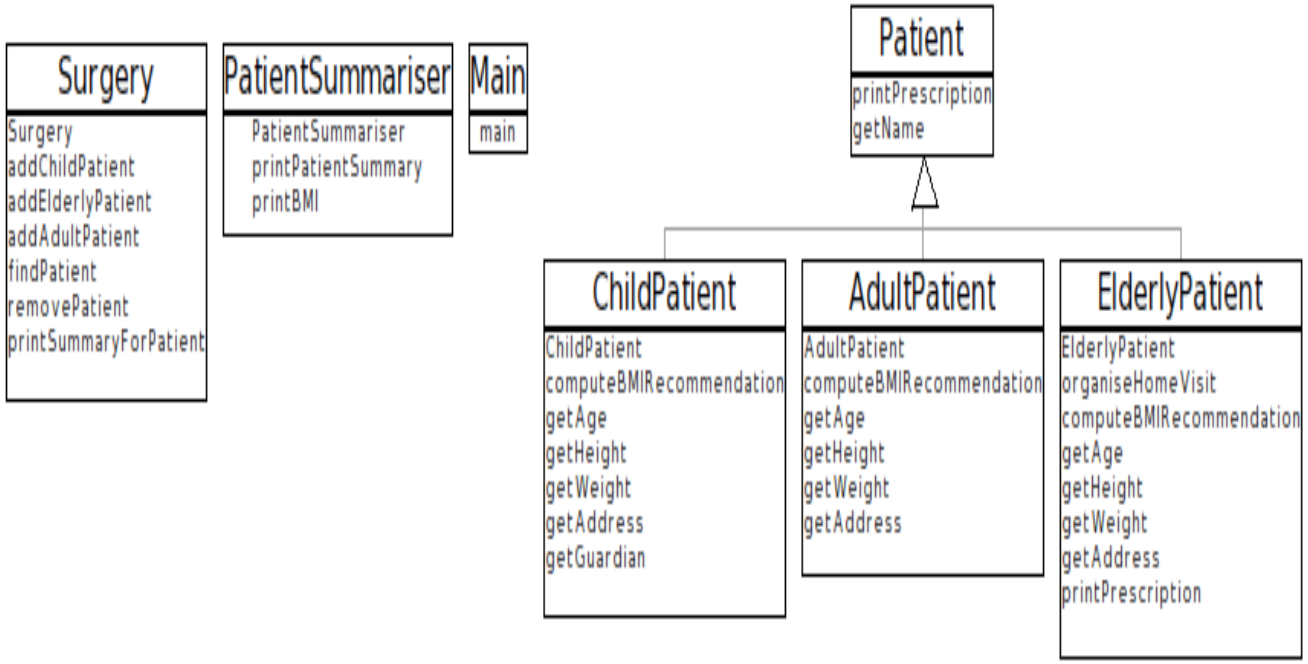


Figure 9.1: Initial Class diagram representing Patient Information System.

including all possible transformation steps and their dependencies among them.

The concrete dependencies/conflicts are depicted in Figure 9.2 as computed from the Petri graph.

The list of all transformations along with their label and unique ids extracted from the approximation are depicted below:

```
List of Transformations/Rules
================================
_895249 | Probe1+
_895250 | Probe1-
_895248 | Probe2+
_895247 | Probe2-
_895246 | Move Method3
_895245 | Pullup Method
_895241 | EncapsulateAttribute
_895243 | Move Method1
_895244 | Pullup Method
_895242 | Pullup Method
_895240 | Pullup Method
_895239 | ExtractSuperclass
================================
```

Search-based Refactoring: The Petri graph serves as input to our search problem. Using the formulation as shown in Section 6.2, we set the construction graph by representing the set of proposed refactorings as nodes while the edges will correspond to dependencies/conflicts between these refactor-

Dependencies

	_895239	_895240	_895241	_895242	_895243	_895244	_895245	_895246	_895247	_895248	_895249	_895250
_895239	X	<	ND	<	ND	<	<	ND	ND	ND	ND	ND
_895240	>	X	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND
_895241	ND	ND	X	ND	ND	ND	ND	ND	ND	ND	ND	ND
_895242	>	ND	ND	X	ND	ND	ND	ND	ND	ND	ND	ND
_895243	ND	ND	ND	ND	X	ND	ND	ND	ND	ND	ND	ND
_895244	>	ND	ND	ND	ND	X	ND	ND	ND	ND	ND	ND
_895245	>	ND	ND	ND	ND	ND	X	ND	ND	ND	ND	ND
_895246	ND	ND	ND	ND	ND	ND	ND	X	ND	ND	ND	ND
_895247	ND	ND	ND	ND	ND	ND	ND	ND	X	ND	ND	ND
_895248	ND	ND	ND	ND	ND	ND	ND	ND	ND	X	ND	ND
_895249	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND	X	ND
_895250	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND	X

Figure 9.2: Dependency table. x denotes mutual exclusion, ND denotes "no dependency" and > shows causality between transformations.

ings derived from the unfolding as shown in Figure 9.2. The construction graph will aid in summarising all interactions among these refactorings and provides input to the search space of the algorithm.

In order to generate solutions from the search space, we choose the initially independent transitions from the Petri graph, as mentioned below:

```
Initially Independent Transformations/Rules:
================================
_895249
_895250
_895248
```

```
_895247
_895246
_895243
_895241
_895239
```

Each transformation will be assigned an identification number in the search space for reference as mentioned below:

```
AntColonySystem for Refactoring:
Ants: 5
Nodes: 11
Iterations: 10
Repetitions: 3
NodeID 0  _895249
NodeID 1  _895250
NodeID 2  _895248
NodeID 3  _895247
NodeID 4  _895246
NodeID 5  _895245
NodeID 6  _895241
NodeID 7  _895243
NodeID 8  _895244
NodeID 9  _895242
NodeID 10 _895240
NodeID 11 _895239
```

In the search space of our algorithm, we employ five ants, each to start with an initially independent transformation and compute its own sequence. They select enabled transformations to move in the search space. This will enlarge their solutions and enable more transformations until all remaining

transformations are in conflict. We derive the following sequences along with the corresponding probe vectors representing the pattern's occurrences, which allow the objective function to evaluate each sequence. In this example we get the same probe vectors for all sequences, hence the objective function $O(s)$ declares the first available candidate solution as an optimal sequence.

The best path computed by the algorithm, representing an optimal sequence of refactorings, is shown below. It represents a sequence of transformations for the initial class model in Figure 9.1. The final optimal sequence is verified and it exists in the real model. The resulting class model is visualised in Figure 9.5.

```
AntID 1, Path=>  [1,11,9,8,7,0,5,10,4,3,2,6]  [1,-1,1,-2]
AntID 2, Path=>  [6,11,9,8,5,7,0,4,10,2,3,1]  [1,-1,1,-2]
AntID 3, Path=>  [3,7,11,0,10,5,4,6,1,2,8,9]  [1,-1,1,-2]
AntID 4, Path=>  [11,3,2,6,10,9,8,7,0,5,4,1]  [1,-1,1,-2]
AntID 5, Path=>  [3,11,7,10,9,8,6,1,5,0,4,2]  [1,-1,1,-2]
Best Path NodeIDs: [1,11,9,8,7,0,5,10,4,3,2, 6]

Best Path RuleIDs:
[_895250 _895239 _895242 _895244 _895243 _895249 _895245
_895240 _895246 _895247 _895248 _895241]
```

Figure 9.3 and 9.4 give an overview of the system metrics before and after the application of the suggested refactoring sequence. The comparison shows that the cyclomatic complexity of the system has been improved and also the number of methods as well as lines of code (LOC) has been reduced in

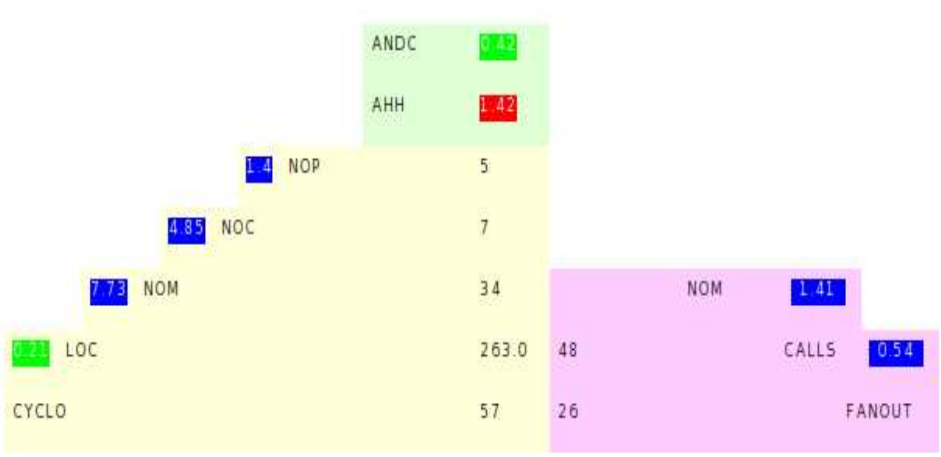


Figure 9.3: Overview pyramid for cyclomatic complexity before refactoring

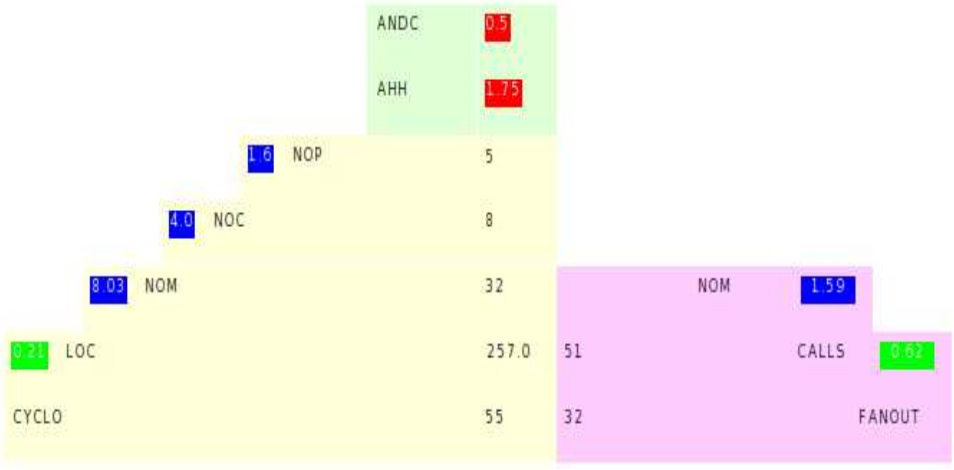


Figure 9.4: Overview pyramid for cyclomatic complexity after refactoring

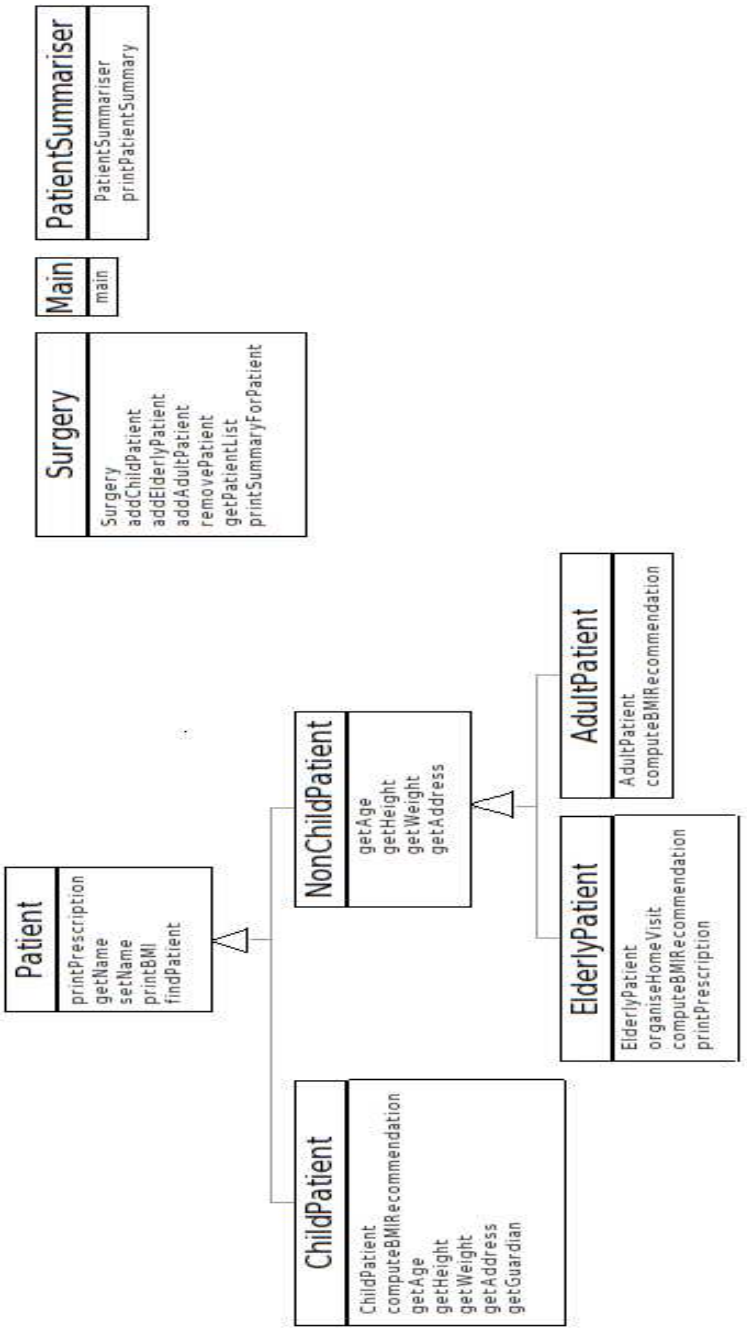


Figure 9.5: Final Class diagram representing Patient Information System.

the refactored system. The abbreviations used in the overview pyramids depicted in Figures 9.3, 9.4 and 9.8 are explained below:

ANDC: Average Number of Derived Classes
AHH: Average Hierarchy Height
NOP: Number of Packages
NOC: Number of Classes
NOM: Number of methods
LOC: Lines of Code
CYCLO: Cyclomatic Number
CALLS: Number of Invocations
FANOUT: Number of Called Classes

Goal-oriented Refactoring: In order to evaluate the goal-oriented refactoring approach, we consider the same example to find a refactoring sequence which removes one of the flaws identified while optimising the quality metrics used to guide the refactoring. In our example, we find out the design flaws with the help of marker *dup* referring to the duplication of methods in the class diagram of PIS, as shown in the Figure 9.6. Below are the list of transformations along with their label and unique ids taken from the approximation.

```
List of Transformations/Rules
==============================
_895251 | dup
_895249 | Probe1+
_895250 | Probe1-
_895248 | Probe2+
```

```
_895247 | Probe2-
_895246 | Move Method3
_895245 | Pullup Method
_895241 | EncapsulateAttribute
_895243 | Move Method1
_895244 | Pullup Method
_895242 | Pullup Method
_895240 | Pullup Method
_895239 | ExtractSuperclass
================================
```

The resulting unfolding of the grammar consists of hypergraph rules (refactoring operations) and the initial PIS design leading to a partial order structure, which is used to find a refactoring sequence in order to remove the flaw identified in Figure 9.6. Each solution starts with a design flaw marked by *dup*, selecting conflicting transformation which will be the final step in the solution removing a flaw. The dependency information obtained from the unfolding is depicted in the Figure 9.7.

The sequence is completed by computing the closure under causal dependencies. In our example, the closure of transformation $id : 895240$ (the step removing the flaw labelled *dup* in the graph of Figure 9.6) under dependencies yields the set {_895239,_895240}. This final sequence also exists in the real model, hence no need of refinement. The resulting design after removing the identified design flaw and its corresponding overview of cyclomatic complexity are shown in Figure 9.9 and 9.8 respectively.

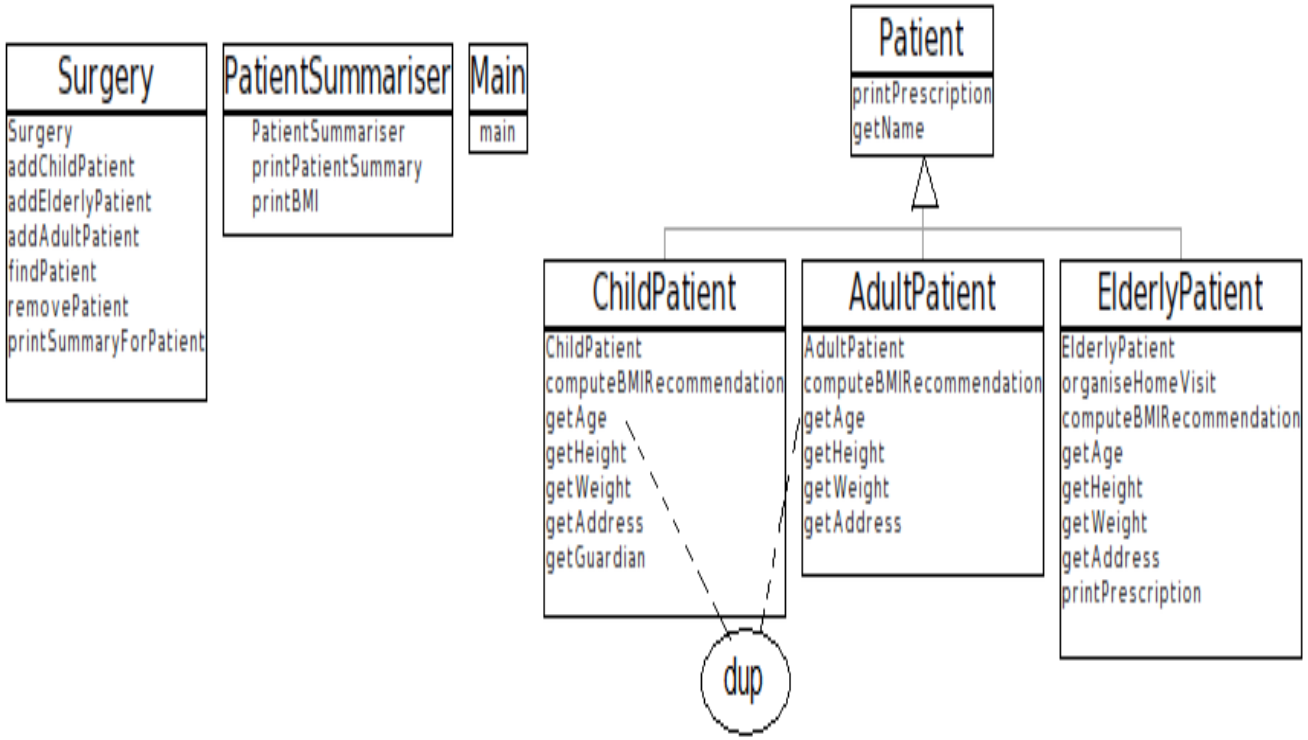


Figure 9.6: Initial Class diagram representing Patient Information System with marker.

9.2 Discussion and Evaluation

We use the unfolding as a scalable representation where designs (solutions) are given by sets of transformations closed under causal dependencies, instead of an explicit representation of the search space of designs and refactorings. We can thus reconstruct solutions when needed, for example in order to evaluate the objective function, but will deal with the more compact representation when navigating the search space. In addition, the approximation guarantees that a finite dependency structure is produced even if the actual state space of the system is infinite [53].

In terms of system size, our approach has the ability to apply to larger models but constructing the approximation for a system takes longer, which is a major limitation in our approach. Apart from this the optimisation part is relatively scalable in terms of handling large sets of refactoring operations for the purpose of search based refactoring. In the above case study it took around two hours to construct the approximation, but the optimisation time is 2 sec in a search space of 12 transformations to compute the optimal sequence.

We support a set of refactorings based on the catalogue [35], i.e., Move Method, Pull Up Method, Extract SuperClass, Encapsulate Attribute, which are implemented together. Most other approaches implement refactorings individually for the purpose of design improvement.

Dependencies

	_895239	_895240	_895241	_895242	_895243	_895244	_895245	_895246	_895247	_895248	_895249	_895250	_895250
_895239	X	<	ND	<	ND	<	<	ND	ND	ND	ND	ND	ND
_895240	>	X	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND
_895241	ND	ND	X	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND
_895242	>	ND	ND	X	ND	ND	ND	ND	ND	ND	ND	ND	ND
_895243	ND	ND	ND	ND	X	ND	ND	ND	ND	ND	ND	ND	ND
_895244	>	ND	ND	ND	ND	X	ND	ND	ND	ND	ND	ND	ND
_895245	>	ND	ND	ND	ND	ND	X	ND	ND	ND	ND	ND	ND
_895246	ND	ND	ND	ND	ND	ND	ND	X	ND	ND	ND	ND	ND
_895247	ND	ND	ND	ND	ND	ND	ND	ND	X	ND	ND	ND	ND
_895248	ND	ND	ND	ND	ND	ND	ND	ND	ND	X	ND	ND	ND
_895249	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND	X	ND	ND
_895250	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND	X	ND
_895251	ND	#	ND	ND	ND	ND	ND	ND	ND	ND	ND	ND	X

Figure 9.7: Dependency table. x denotes mutual exclusion, ND denotes “no dependency”, > shows causality and # denotes conflict between transformations.

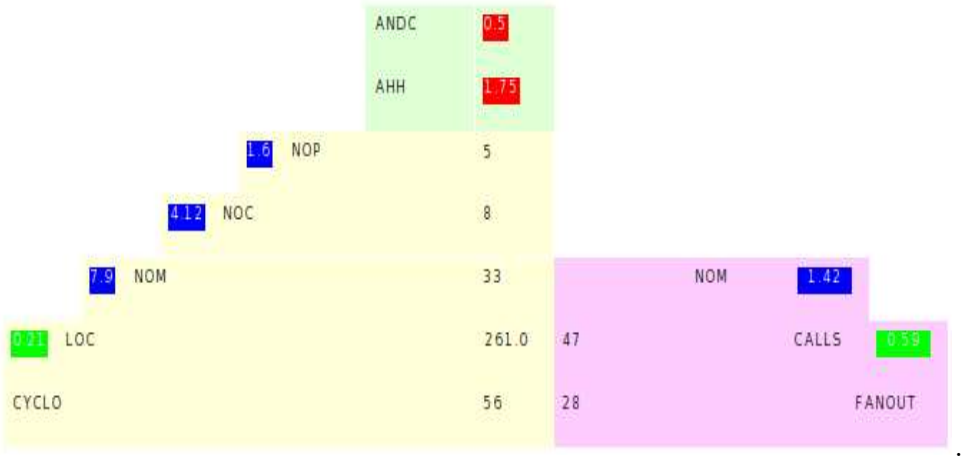


Figure 9.8: Overview pyramid for cyclomatic complexity after refactoring

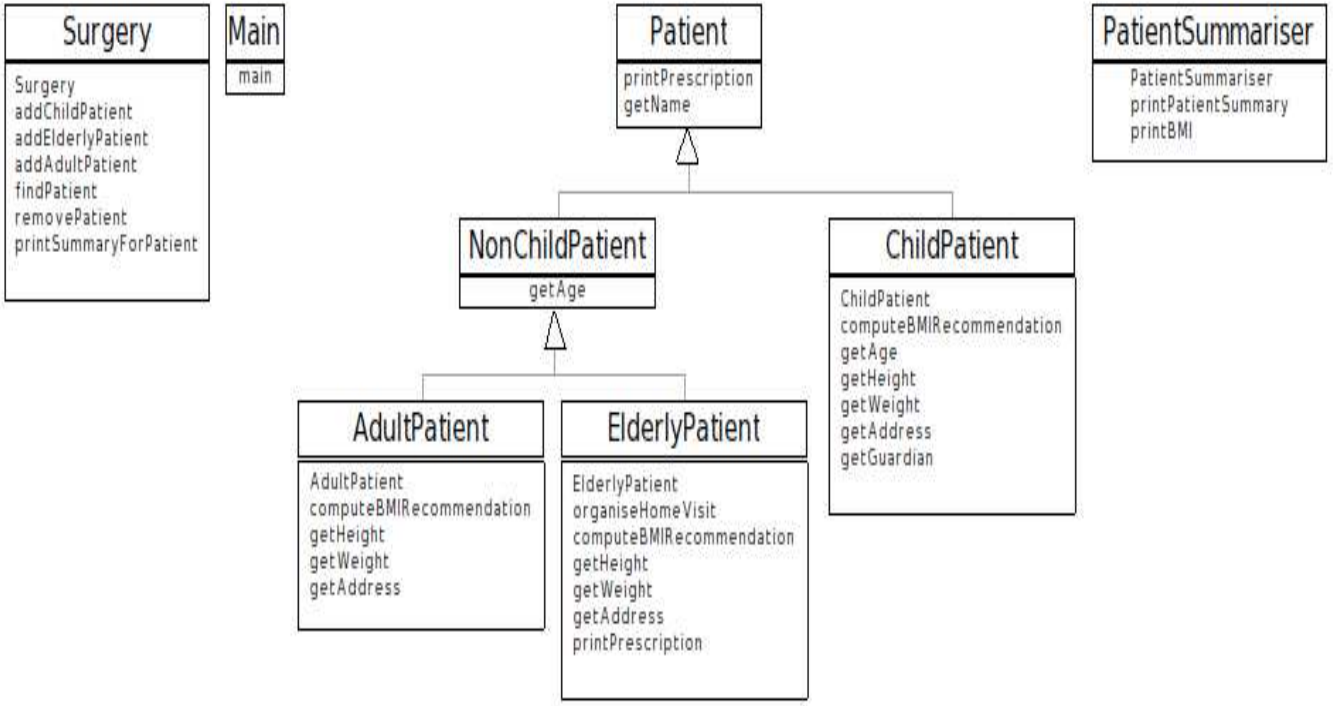


Figure 9.9: Initial Class diagram representing Patient Information System after refactoring.

Chapter 10

Conclusion and Future Work

In this thesis, we are primarily interested in studying how the formulation of refactoring as graph rewriting can be exploited for optimisation. The presented approach involves a combination of graph transformation theory and the ACO metaheuristic, aiming to implement search-based refactoring. Hence the technique relies on a formalisation of refactoring based on graph transformation [61] with the analysis of dependencies extracted from their approximated unfolding [6] by means of an optimisation using the ACO [28].

To improve traceability/understandability, we are using the causal relation to explain refactorings in terms of their interdependency. For example, if the programmer accepts that the last step performed represents an improvement, they will implicitly accept the relevance of all changes up to that point that the final step depends on. We can also specify constraints on the sets of

transformations to ensure that every step contributes directly or indirectly to the last step in the sequence. The use of dependency information between transformations allows us to remove steps that are unrelated to the intended change, making each change relevant and therefore easier to interpret. An evaluation of understandability through experiments with smaller models, assessing the effort it takes a human developer to understand the changes proposed by the search-based approach, is providing one of the interesting challenge in the future.

It seems apparent that goal-oriented refactorings are easier to understand than those obtained by global optimisation while at the same time leaving control with the developer through selecting the goal in terms of the design flaws to be removed. An experimental evaluation of usability is a topic for future work.

AUGUR2 was developed as a prototype tool and used for the analysis of graph transformation systems. Currently, it doesn't allow us to create approximations for larger examples. Hence, we are interested to improve the matching algorithm in AUGUR2 in order to produce approximations for larger models.

Obviously, there are some options to increase efficiency (i.e., no redundancy and no coverability check, etc.) but still the runtime is long for larger graphs. In AUGUR2, we tried to use the search plan algorithm (see Op-

tions), which is a more efficient implementation of the matching algorithm, but it did not seem to help very much. Therefore, the optimisation of the matching algorithm in AUGUR2 is essential future work.

We have implemented the approach up to a point where we can check for smaller graphs that the sequences produced in the approximated model are also executable in the full model. The verification of the sequence for larger models will be future work.

Bibliography

[1] P. Abdulla, B. Jonsson, M. Kindahl, and D. Peled. A general approach to partial order reductions in symbolic verification. In *Computer Aided Verification*, pages 379–390. Springer, 1998.

[2] J.S. Aguilar-Ruiz, I. Ramos, J.C. Riquelme, and M. Toro. An evolutionary approach to estimating software development projects 1. *Information and Software Technology*, 43(14):875–882, 2001.

[3] G. Antoniol, M. Di Penta, and M. Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 240–249, Washington, DC, USA, 2005. IEEE Computer Society.

[4] D. Astels. Refactoring with UML. In *3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002)*, pages 67–70, 2002.

[5] A.J. Bagnall, V.J. Rayward-Smith, and I.M. Whittley. The next release problem. *Information and Software Technology*, vol:43(14):pages 883–890, 2001.

[6] P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, pages 381–395. Springer-Verlag, 2001. LNCS 2154.

[7] P. Baldan, A. Corradini, and B. König. Static analysis of distributed systems with mobility specified by graph grammars—a case study. In *Proc. of IDPT '02 (Sixth International Conference on Integrated Design & Process Technology)*. Society for Design and Process Science, 2002.

[8] P. Baldan, A. Corradini, and U. Montanari. Unfolding and event structure semantics for graph grammars. In *FoSSaCS '99: Held as Part of the European Joint Conference on the Theory and Practice of Software, ETAPS'99*, pages 73–89, London, UK, 1999. Springer-Verlag.

[9] P. Baldan, A. Corradini, and U. Montanari. Contextual petri nets, asymmetric event structures, and processes. *Information and Computation*, 171(1):1–49, 2001.

[10] P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In *ICGT*, pages LNCS vol:2505,14–29, 2002. Springer-

Verlag.

[11] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1329–1336, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[12] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, chapter 30, pages 425–439. Springer Berlin/Heidelberg, 2006.

[13] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35:268–308, September 2003.

[14] T. Bodhuin, G. Canfora, and L. Troiano. Sormasa: A tool for suggesting model refactoring actions by metrics-led genetic algorithm. In *Proceedings of the 1st Workshop on Refactoring Tools (WRT' 07-in conjunction with ECOOP'07)*, pages 23–24, 2007.

[15] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 1885–1892, New York, NY, USA, 2006. ACM.

[16] Lionel C. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 43–50, New York, NY, USA, 2002. ACM.

[17] G. Canfora, M. Di Penta, R. Esposito, and M.L. Villani. An approach for QoS-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1069–1075, New York, NY, USA, 2005. ACM.

[18] U. Chirico. A Java framework for ant colony systems. In *Ants2004: Forth International Workshop on Ant Colony Optimization and Swarm Intelligence, Brussels September 5-8, Proceeding Series: LNCS, Vol. 3172*, 2004.

[19] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages

154–169. 2000, LNCS Springer.

[20] J. Clarke, J.J. Dolado, M. Harman, R.M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *Software, IEEE Proceedings*, 150(3):161 – 175, June 2003.

[21] Keith D. Cooper, Philip J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, LCTES '99, pages 1–9, New York, NY, USA, 1999. ACM.

[22] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation. Part I: Basic concepts and double pushout approach. In *Handbook of graph grammars and computing by graph transformation*, pages 163–245. World Scientific Publishing Co., Inc., 1997.

[23] S. Demeyer, F. Van Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. Du Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall, and M. El-Ramly. The LAN-simulation: a refactoring teach-

ing example. In *Principles of Software Evolution, Eighth International Workshop on*, pages 123 – 131, sept. 2005.

[24] A. Deursen, LMF Moonen, A. Bergh, and G. Kok. Refactoring test code. *CWI. Software Engineering [SEN]*, (R 0119):1–6, 2001.

[25] M. Dorigo and G. Di Caro. *The ant colony optimization meta-heuristic*, pages 11–32. New ideas in optimization, McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999.

[26] M. Dorigo and L.M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[27] M Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy. *Technical Report 91016*, 91-016(91016), 1991.

[28] M. Dorigo and T. Stützle. *Ant colony optimization*. MIT Press, 2004.

[29] F.L. Dotti, B. König, O. Marchi dos Santos, and L. Ribeiro. A case study: Verifying a mutual exclusion protocol with process creation using graph transformation systems. Technical Report 08/2004, Universität Stuttgart, 2004.

[30] S. Ducasse, M. Lanza, and S. Tichelaar. The MOOSE reengineering environment. *Smalltalk Chronicles*, 3(2), 2001.

[31] H. Ehrig. Tutorial introduction to the algebraic approach of graph grammars. In *Graph-Grammars and Their Application to Computer Science: 3rd International Workshop, Warrenton, Virginia, USA, December 2-6, 1986*, volume 291, page 1. Springer Verlag, 1988.

[32] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[33] H. Ehrig, M. Pfender, and H.J. Schneider. Graph-grammars: An algebraic approach. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 167–180. IEEE, 1973.

[34] D. Fatiregun, M. Harman, and R.M. Hierons. Search-based amorphous slicing. In *Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[35] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[36] L.M. Gambardella and M. Dorigo. An ant colony system hybridized with a new local search for the sequential ordering problem. *INFORMS*

J. on Computing, 12(3):237–255, 2000.

[37] R. Gheyi, T. Massoni, and P. Borba. Type-safe refactorings for Alloy. In *Proceedings 8th Brazilian Symposium on Formal Methods*, pages 174–190, 2005.

[38] M. Harman and J. Clark. Metrics are fitness functions too. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 58–69. IEEE, 2004.

[39] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Trans. Softw. Eng.*, 30:3–16, January 2004.

[40] M. Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839, 2001.

[41] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *GECCO 2007 (Conf. on Genetic and evolutionary computation)*, pages 1106–1113, New York, NY, USA, 2007. ACM.

[42] M. Harman and J. Wegener. Getting results from search-based approaches to software engineering. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 728–729, Washington, DC, USA, 2004. IEEE Computer Society.

[43] R. Heckel, J.M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *Graph transformation: first international conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002: proceedings*, volume 2505, page 161. Springer Verlag, 2002.

[44] A. Hilgers and BJ Boersma. Optimization of turbulent jet mixing. *Fluid dynamics research*, 29(6):345–368, 2001.

[45] R.C. Holt, A. Schurr, S.E. Sim, and A. Winter. GXL: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, 2006.

[46] R.C. Holt, A. Winter, and A. Schürr. Gxl: Toward a standard exchange format. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 162–171. IEEE, 2000.

[47] inFusion http://www.intooitus.com/products/infusion, Dated: 11-09-2011.

[48] R. Johnson, E. Gamma, R. Helm, and J. Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1(1-2):33–57, 1995.

[49] C. Kaner, S. Member, and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *In METRICS 2004. IEEE CS*. Press, 2004.

[50] J. Kerievsky. *Refactoring to patterns*. Pearson Education, 2005.

[51] B. König and V. Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Proc. of TACAS '06*, pages 197–211. Springer, 2006. LNCS 3920.

[52] B. König and V. Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. Technical report, Universität Stuttgart, 2006.

[53] B. König and V. Kozioura. AUGUR2—a new version of a tool for the analysis of graph transformation systems. In *Proc. of GT-VMT '06 (Workshop on Graph Transformation and Visual Modeling Techniques)*, volume 211 of *ENTCS*, pages 201–210. Elsevier, 2008.

[54] V. Kozyura. Abstraction and abstraction refinement in the verification of graph transformation systems. 2010. http://duepublico.uni-duisburg-essen.de/servlets/DocumentServlet?id=21627.

[55] H.J. Kreowski. A comparison between petri-nets and graph grammars. *Graphtheoretic Concepts in Computer Science*, pages 306–317, 1981, Springer.

[56] A. Kuhn and T. Verwaest. Fame, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, pages 57–66. Citeseer, 2008.

[57] L. Lambers. A new version of GTXL: An exchange format for graph transformation systems. *In Proc. Workshop on Graph-Based Tools (GraBaTs' 04*.

[58] Z. Li, M. Harman, and R.M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33:225–237, 2007.

[59] T. Mens. On the use of graph transformations for model refactoring. *Generative and transformational techniques in software engineering*, pages LNCS, 2006, Volume 4143/2006, 219–257, 2006.

[60] T. Mens, G. Taentzer, and D. Müller. Challenges in model refactoring. In *Proc. 1st Workshop on Refactoring Tools, University of Berlin*, volume 98, 2007.

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

[62] T. Mens, R. Van Der Straeten, and J.F. Warny. Graph-based tool support to improve model quality. In *Workshop on Quality in Modeling*, page 47, 2006. Co-located with the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems, 2006.

[63] Brian S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1375–1382, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[64] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, Part I. *Theoretical Computer Science*, 13(1):85–108, 1981.

[65] M. O'Keeffe and M. Ó Cinnéide. A stochastic approach to automated design improvement. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, PPPJ '03, pages 59–62, New York, NY, USA, 2003. Computer Science Press, Inc.

[66] M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.*, 20(5):345–364, 2008.

[67] M. O'Keeffe and M. O'Cinneide. Search-based software maintenance. In *Proceedings of the Conference on Software Maintenance and Reengineering*, pages 249–260, Washington, DC, USA, 2006. IEEE Computer Society.

[68] OMG. Unified Modeling Language: Superstructure version 2.0. formal/2005-07-04, August 2005.

[69] J. Otamendi. The importance of the objective function definition and evaluation in the performance of the search algorithms. Proceedings of the 16TH European Simulation Symposium, 2004, Budapest, Hungary.

[70] F. Qayum. Automated assistance for search-based refactoring using unfolding of graph transformation systems. In *Proceedings of the 5th international conference on Graph transformations*, ICGT'10, pages LNCS, 2010, Volume 6372/2010, 407–409, Berlin, Heidelberg, 2010. Springer-Verlag.

[71] F. Qayum and R. Heckel. Analysing refactoring dependencies using unfolding of graph transformation systems. In *Proceedings of the 7th*

International Conference on Frontiers of Information Technology, FIT '09, pages 15:1–15:5, New York, NY, USA, 2009. ACM.

[72] F. Qayum and R. Heckel. Local search-based refactoring as graph transformation. In *Proceedings of the 2009 1st International Symposium on Search Based Software Engineering*, SSBSE '09, pages 43–46, Washington, DC, USA, 2009. IEEE Computer Society.

[73] F. Qayum and R. Heckel. Search-based refactoring using unfolding of graph transformation systems. *International Conference on Graph Transformation 2010 - Doctoral Symposium, Electronic Communications of the EASST*, 38(0), 2011.

[74] G. Rangel, L. Lambers, B. Konig, H. Ehrig, and P. Baldan. Behavior preservation in model refactoring using dpo transformations with borrowed contexts. In *Graph transformations: 4th international conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008: proceedings*, volume 5214, page 242. Springer-Verlag New York Inc, 2008.

[75] W. Reisig. *Petri nets: An introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[76] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Sys-*

tems (TOPLAS), 24(3):217–298, 2002.

[77] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO 2006*, pages 1909–1916, New York, NY, USA, 2006. ACM.

[78] G. Sunyé, D. Pollet, Y. Le Traon, and J.M. Jézéquel. Refactoring UML models. *«UML» The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 134–148, 2001.

[79] G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 44(4):28 – 40, 2001. UNIGRA 2001, Uniform Approaches to Graphical Process Specification Techniques (a Satellite Event of ETAPS 2001).

[80] L. Tahvildar and K. Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(4-5):331–361, 2004.

[81] R. Van Der Straeten and M. D'Hondt. Model refactorings through rule-based inconsistency resolution. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1210–1217. ACM, 2006.

[82] R. Van Der Straeten, V. Jonckers, and T. Mens. Supporting model refactorings through behaviour inheritance consistencies. *The Unified Modeling Language. Modelling Languages and Applications, LNCS Volume:3273,*, pages 305–319, 2004, Springer-Verlag.

[83] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002.

[84] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. *«UML» 2003-The Unified Modeling Language. Modeling Languages and Applications, 6th international conference, San Francisco, CA, USA, October 20-24, 2003: proceedings. LNCS, volume:2863, Springer-Verlag.*, pages 144–158, 2003.

[85] A. Winter. Exchanging graphs with GXL. *Graph Drawing- 9th International Symposium proceedings. Vienna, Austria, September 23-26, 2001*, Volume:9:485, 2002, Springer-Verlag,.

[86] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, 2002. Springer-Verlag.

[87] J. Zhang, Y. Lin, and J. Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Volume II of Research and Practice in Software Engineering*, pages 199–218. Springer, 2005.