A Program Transformation Step Prediction based Reengineering Approach

Ph.D. Thesis

Shaoyun Li

Software Technology Research Laboratory

De Montfort University

2007

To *my husband*, Kaixin Zhou and

my parents and *parents-in-law*

for their love and support

Declaration

I declare that the work described in this thesis was originally carried out by me during the period of registration for the degree of Doctor of Philosophy at De Montfort University, U.K., from December 2002 to December 2005. It is submitted for the degree of Doctor of Philosophy at De Montfort University. Apart from the degree that this thesis is currently applying for, no other academic degree or award was applied for by me based on this work.

Acknowledgement

First of all, I wish to acknowledge the financial support from De Montfort University for the research work.

My deepest gratitude goes to my supervisor, Professor Hongji Yang. He is the one who brought me numerous chances, unfaltering patience, meticulous and inspiring advice and equipped me with the courage to strive through one of the most challenging periods of my life yet make it one of my most important times as well. I am grateful for his leading role fostering my academic, professional and personal growth.

I would like to express my deep appreciation to my second supervisor Dr. Martin Ward, for his invaluable advice and constant support. I am sincerely grateful to Professor Hussein Zedan for his precious comments and suggestions about my work, which means a lot to me. I appreciate Dr. Samad Ahmadi for his help with my work on various occasions. I feel so blessed to have many discussions with them.

I sincerely thank Professor Hua Zhou for encouraging me to pursue my PhD degree. I am grateful to my fellows at Yunnan University in China, Professor Hongzhi Liao, Professor Tong Li, Qing Duan and the rest members of Information Technology Institute.

A great many thanks go to the colleagues at De Montfort University, Feng Chen, Clinton Ingrams, Monika Solanki, Zhuopeng Zhang, Matthias Ladkau, Stefan Natelberg and many others. I thank them for their help and encouragement during the past years.

Finally, I am particularly grateful to my husband, Kaixin Zhou, my parents and parents-in-law from the bottom of my heart – no words are adequate to express my appreciation for their love for all my life. Their love empowers me to steer through ups and downs and helped me reach this point today. This thesis is dedicated to them.

Abstract

The essence of software reengineering is to improve or transform existing software so that it can be understood, controlled and used anew. Program transformation is used as a core technique for fulfilling the various needs in the context of software reengineering. The improvement of the automation and efficiency of program transformations for reengineering is a concern in both research and industrial areas. The proposed research aims to achieve the goal by providing an appropriate mechanism to predict the transformation steps to fulfil specific reengineering targets to enhance the efficiency and correctness of reengineering through program transformations.

In this thesis, a Target Driven Program Transformation Step Prediction approach (TDPTSP) is proposed to assist the process of transformation in software reengineering. The proposed approach is explored by using a transformation-intensive language Wide Spectrum Language (WSL) as an intermediate language and its toolset that provides a well-developed transformation bank containing a large number of proven transformations. The predication of transformations is an intelligent means to guide the transformation process towards reengineering targets. In order to make the identified targets tangible, the concept of Target Model (TM) is introduced for the target representation and evaluation. In the model, software metrics selected from a reengineering intensive metrics catalogue are correlated to the corresponding targets.

With the quantitative measurement and the tangible target representation, the program transformation step prediction algorithm is constructed as a heuristic based search approach. Expertise for applying program transformations in the practical work is essential for the prediction operation. The prediction approach incorporates the expertise rules in addition to the metrics based approach. When predicting the transformations on domain specific applications, domain features are the vital factors. Therefore, the

approach needs to be augmented to deal with such applications. To explore how to utilise the transformation prediction for the applications in specific domain, multimedia domain is chosen for the study. In order to exploit the proposed approach, WSL is extended with object-oriented features and multimedia domain features consistently based on the existing language levels of WSL. Correspondingly, the existing transformation bank is extended for the needs of the transformation prediction driven by reengineering targets. A prototype tool and three case studies are presented for the experiments to show the proposed approach is feasible and promising. Conclusions are drawn based on analysis and further research directions are discussed at the end of the thesis.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ABST-CFDF	Abstractness based on Control-Flow Data-Flow Complexity metric
ABST-LOC	Abstractness based on Lines of Code metric
ABST-STAT	Abstractness in Statement metric
ABST-VOC	Abstractness in Vocabulary metric
AI	Artificial Intelligent
AMS	Average Module Size metric
ANFC	Average Number of Feature implemented in a Class metric
ANSDF	Average Number of Shared Data module between two Features metric
ANSFF	Average Number of Shared Functions between two Features metric
AST	Abstract Syntax Tree
BNF	Backus Naur Form
CBO	Coupling Between Object Classes metric
CF	Control Flow
CFDF	Control Flow and Data Flow metric
DBMS	Database Management System
DF	Data Flow
DIT	Depth of Inheritance Tree metric
ETL	Error Tolerance Level metric
F-DOC	FIP Documentation Tool
FIP	FermaT Integration Platform
F-ME	FIP Maintenance Environment
FOPF	First-Order Predicate Formulas
F-TP	FIP Transformation Predictor
F-UML	FIP UML
HIL	Human Interaction Level metric
HLL	High Level Language
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organisation for Standardisation
JavaCC	Java Compiler Compiler
LDA	Linkage Disequilibrium Analyser

MA	Maintainer's Assistant
MOTMET	Model of Target-Metric-Transformation
NCIF	Number of Classes Implementing a Feature metric
NCNB	Non-Comment Non-Blank metric
NEI	Number of External Interactions metric
NFR	Non-Functional Requirements
NMI	Number of Method Invocation metric
NON	Number of Node metric
NVC	Number of Variables per Class metric
OO	Object Oriented
OSC	Overlap Statements among the Classes metric
PSR	Percentage of Spatial Constructs metric
PTR	Percentage of Temporal Constructs metric
RNC	Recursion and Nesting metric
SBSE	Search Based Software Engineering
SD	Self-Descriptiveness metric
SMIL	Synchronised Multimedia Integration Language
TDPTSP	Target Driven Program Transformation Steps Prediction
TDTPF	Target Driven Transformation Prediction Framework
TM	Target Model
TP	Transformation Path
TPM	Transformation Process Model
TQS	Transformation Qualification Score
TS	Target Score
TSG	Transformation Search Graph
WMC	Weighted Methods per Class metric
WOC	Weight Of every Construct metric
WOIL	Weight Of Interfaces in relations to Lines of code metric
WP	Weakest Precondition
WSL	Wide Spectrum Language
XML	The Extensible Markup Language
XSLT	The Extensible Stylesheet Language Transformation

Chapter 1
Introduction

Objectives

- To observe the need for reengineering target driven program transformation prediction
- To present the scope of the thesis
- To highlight original contributions and define the research questions
- To outline the organisation of the thesis

1.1 Motivation and Targets of Research

An intrinsic property of software in a real-world environment is that software systems are continuously being evolved soon after their first version is delivered to meet the changing requirements of their users. Software practitioners are performing changes daily to source code such as correcting errors, adding new functionalities and adopting new technologies to ensure the users' needs and the changing environment are met [68].

Software reengineering is often viewed as an attractive approach for such evolution and has emerged as a business critical activity over the past decade. Most reengineering

methodologies have come to rely extensively on tools in order to reduce human effort requirements. Not surprisingly, the topic of software reengineering has been researched heavily for some time, leading both to a variety of commercial toolsets for particular reengineering tasks and to research prototypes [103]. Each task taken in software reengineering process is driven by its specific target(s).

Reengineering consists of mainly two parts, reverse engineering and forward engineering through which an existing system is systematically transformed into a new form to fulfil the needs of software evolution [8]. The purpose of software reengineering is to realise quality improvements in operation, system capability, functionality, performance, evolvability at a lower cost, schedule, or risk to the customer [12]. A reengineering process can start from the source code that is the most reliable information of software and end at a desired form of the software. The targeted form can not only be the implementation of the system harnessed with new technology, improved quality or needed new functionalities, but also be the specification derived from the source code which are used to comprehend or analyse the system and accordingly facilitate the forward engineering.

In the context of software reengineering, program transformation is used, which is the examination and alteration of one representation to reconstitute it in a new form and the subsequent implementation of the new form while preserving the subject system's external behaviour (functionality and semantics) [24]. Program transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system. Program transformation is also needed to convert legacy code or deteriorated code into a more modular or

structured form [39] or even to migrate code to a different programming language or even language paradigm [40]. To summarise, program transformation is a crucial means to realise a variety of software reengineering tasks driven by specific needs or targets.

However, there are a number of problems caused in the applications of program transformations for reengineering. (1) In most cases, the corresponding relation between reengineering target and program transformation that contributes the satisficing the target is not explicitly one-to-one. In order to achieve a reengineering target, there might be more than one transformation as candidates needed. (2) Within the transformation candidates, different one can result in the outcome with different satisficed degrees. (3) The different execution sequence of transformations can cause different impacts. (4) Normally, to make the two decisions, i.e. selecting transformations in the existing transformation bank and deciding the execution sequence, relies on the features of the source code, the capability of the program transformations and the experience of the software engineer. Doing the work without the experience could cause the lack of efficiency and correctness to fulfil the reengineering target(s). (5) When processing a domain specific application, the domain features are important factors for determining transformations due to their speciality. They are unavoidable factors taken into account when the domain specific applications are reengineered.

The thesis therefore aims to present a program transformation step prediction method driven by reengineering target with quantitative means, the knowledge of expertise and domain features as a solution to address the above five problems.

1.2 Scope of Thesis

In this thesis, a program transformation step prediction approach for software reengineering is proposed. The approach is based on the construction of a Wide

Spectrum Language (WSL) and its program transformation theory [125]. The thesis concentrates on the prediction method of program transformations for identified reengineering targets and studies the proposed approach on the normal imperative or object-oriented programs as well as the programs in special domains, where multimedia domain is selected. The scope of the research includes:

(1) Using WSL as an intermediate language and its toolset to explore the proposed approach. Because WSL represents specifications and executable implementations, it is ideal for reengineering purposes. A well-developed library of proven transformations based on WSL can support the research on the program transformation prediction.

(2) Proposing the idea of transformation prediction. The predication of transformations is an intelligent means to guide the transformation process towards the reengineering targets. To provide the useful information guiding the transformation process, the transformation engine is supposed to determine the suitable candidates and predict the sequence of the transformations. This feature of the transformation tools can assist and present clues to the users and accordingly improve the transformation implementation's efficiency.

(3) Presenting the target model to represent and measure reengineering target. In order to make the identified targets tangible, a formal representation is necessary. The goal driven model [89] is used for the representation of the targets. The software metrics related to a target are included in the model.

(4) Using software metrics to measure the status of program and satisficed degree of the target based on the target model. Software metrics are useful quantitative means to measure the status of software and the satisficed degree of targets. Six

groups of dedicated reengineering intensive metrics are selected and counted based on the target model.

(5) Extending WSL with object-oriented features so that the scope of the WSL's application is broadened to particular domains subsequently. Based on the hierarchy of WSL constructs and the fixed-point theory, the object-oriented syntax and semantics are consistent with the original WSL, therefore the transformations developed based on the extended WSL can be compatible with the existing transformations and vice versa.

(6) Extending WSL with domain specific features. The extension is applied in the multimedia domain by extending the language and mapping the application of the domain into WSL.

(7) Extending the current transformation bank for the transformation prediction purpose. The extension works on two aspects, one is the management of the transformation bank and the other is the addition of the transformations based on the extended WSL. The transformations are classified according to their characteristics and usage. The classification will be used as a heuristic for the transformation prediction algorithm.

(8) Using a search-based approach to constructing transformation process model that is used to predict program transformation candidates and their execution sequence. With the quantitative measurement and the tangible target representation, the transformation prediction can be modelled as a search problem. It may prove that a metrics based prediction algorithm is not efficient due to the large search space. The expertise is an important knowledge for the transformation prediction. To incorporate the expertise formalised as a set of rules

and take the domain features into account will improve the prediction approach.

1.3 Original Contributions

The original contributions of this thesis are listed as follows.

(1) The most significant contribution is to propose the motivation of program transformation step prediction for reengineering, model the transformation prediction as a search problem and provide the heuristic based algorithms as solutions. By studying the source code and using the prediction algorithm, a set of transformation candidates and the possible execution sequence can be predicted to provide the information to software engineer and guide the practical transformation work.

(2) The second contribution is to narrow the large search space by using heuristics. The metrics based approach is proposed with the utilisation of the heuristics. The expertise obtained in the practical work is taken into account to incorporate the prediction process. In addition, the thesis explores the transformation prediction based approach with using domain features. The multimedia domain is chosen for the investigation.

(3) The third contribution is to model the reengineering targets and measure them with the dedicated software metrics. Software reengineering is composed of various tasks for different purposes related to software evolution. Program transformation has been used as a practical technique to realise these tasks. To model the task as target-driven is natural because each task has its target. In the research described in the thesis, the program transformations are regarded as reengineering target-driven to implement the required reengineering tasks. The targets are modelled by using the goal-driven techniques and correlated the

reengineering intensive metrics to the targets. Doing so can provide a tangible way to guide the program transformation process towards the given targets.

(4) The fourth contribution is to extend WSL. The transformation-intensive WSL is used as an intermediate language to experiment the program transformation theory and application. However, the current transformations are developed for procedural languages only. In order to experiment the proposed approach in more cases, it is necessary to extend the existing language and transformations. In the thesis, the WSL is extended with object-oriented features and the multimedia domain features. In addition, the *Meta*WSL is also extended with more functions for the transformation prediction.

(5) The fifth contribution is to extend the transformation bank. As the WSL is extended with the advanced features, the transformations should be extended accordingly. On the other hand, the transformation bank is also structured for the prediction particularly.

(6) The sixth contribution, which is quite novel, is to apply program transformations on multimedia applications for reengineering purposes.

1.4 Research Questions

The whole research question is how well the proposed research supports software reengineering. The following specific research questions are given to judge the success of the research described in this thesis:

- What are the advantages of the implementation of reengineering with the program transformation prediction approach against the one without using the approach?

- Are the extended constructs consistent with the syntax and semantics of WSL?
- Is the target model correct and complete to represent the identified target?
- Can the prediction be modelled as a search problem?
- What kind heuristics will be used in the proposed approach? Is the heuristic knowledge useful for the transformation prediction?
- How can the quantitative approach be used to control the transformation prediction process?
- Will the prediction result be ensured as a good solution?

1.5 Organisation of Thesis

The rest of the thesis is organised as follows.

Chapter 2 provides an overview of reengineering and program transformation, investigates the existing related work, especially those involving existing program transformation systems, software metrics, software requirement specification and program transformation automation for software reengineering. Particularly, the transformation specific language WSL, the transformation theory and the support tools are reviewed briefly.

Chapter 3 outlines target driven transformation prediction framework. The needs of the proposed research are analysed and the technical steps in the approach are discussed.

Chapter 4 presents and justifies six dedicated groups of reengineering-intensive software metrics and depicts the way to model a reengineering target that can be measured with

the metrics.

Chapter 5 presents the extension of WSL with object-oriented features on both syntax and semantics. In addition, the extension of WSL into the multimedia is explored for the experiment of the proposed approach in a specific domain. Based on the extension of the language, the construction of the transformation bank, which provides a mechanism to manage transformations and control transformation process, is shown.

Chapter 6 illustrates the core technique of the proposed research. The technical steps of the program transformation predication are elaborated. The algorithms of the steps are presented in detail.

Chapter 7 describes the implementation of the prototype developed for the proposed approach and gives three case studies on a procedural program, an object-oriented program and a multimedia application respectively to address the usability of the approach on different aspects.

Chapter 8 summarises the thesis, draws the conclusion and propose the future work. The research questions proposed in this chapter are answered to evaluate the proposed approach.

Appendix A gives the syntax extension of WSL.

Appendix B shows an XML-based representation of target model

Appendix C lists the transformations stored in the transformation bank.

Appendix D lists all the related publications by the author during the PhD study.

Chapter 2
Related Research

Objectives

- To provide the related work of the thesis
- To present the basic concepts related to software reengineering and program transformation
- To discuss existing techniques to modelling software requirements and software metrics
- To review the research on program transformation automation
- To overview the WSL and its program transformation theory

2.1 Introduction

This research aims to provide a method for predicting transformations driven by reengineering targets based on a transformation-specific language WSL. Four major tasks are implied in this statement: (1) using program transformation to implement reengineering; (2) applying the transformations stored in the existing transformation systems; (3) modelling the targets; (4) predicting the suitable transformation steps. This

chapter will review some existing techniques related to the aimed research.

2.2 Software Reengineering

2.2.1 Definition of Software Reengineering

Software reengineering is the examination, analysis and alteration of an existing software system to reconstitute it in a new form and the subsequent implementation of the new form [85]. The process typically encompasses a combination of other processes such as reverse engineering, redocumentation, restructuring, translation and forward engineering. The goal is to understand the existing software (specification, design, implementation) and then to re-implement it to improve the system's functionality, performance or implementation. The objective is to maintain the existing functionality and prepare for new functionality to be added later.

2.2.2 Objectives of Software Reengineering

The challenge in software reengineering is to take existing systems and instill good software development methods and properties, generating a new target system that maintains the required functionality while applying new technologies. Although specific objectives of a reengineering task are determined by the goals of the corporations, there are four general reengineering objectives [96]:

- Preparation for functional enhancement
- Improve maintainability
- Migration
- Improve reliability

Although reengineering should not be done to enhance the functionality of an existing

system, it is often used in preparation for enhancement. Legacy systems, through years of modifications due to errors or enhancements, become difficult and expensive to change. The code no longer has a clear and logical structure, its documentation may not exist, and if it exists, it is often outdated. Reengineering specifies the characteristics of the existing system that can be compared to the specifications of the characteristics of the desired system. The reengineered target system can be built to facilitate easily the enhancements. For example, if the desired system enhancements build on object-oriented design, the target system can be developed using object-oriented technology in preparation for increasing the functionality of the legacy system.

As systems grow and evolve, maintainability costs increase because changes become difficult and time consuming. An objective of reengineering is to re-design the system with more appropriately functional modules and explicit interfaces. Documentation, internal and external, will also be current, hence improving maintainability.

The computer industry continues to grow at a fast rate; new hardware and software systems include new features, quickly outdating current systems. As these systems change, personnel skills migrate to the newer technologies, leaving fewer people to maintain the older systems. In a relatively short time, manufacturers no longer support the software and hardware parts become expensive. Even more important is the compatibility of the older systems with the newer ones. For these reasons, companies with working software that meets their needs might need to migrate to a newer hardware platform, operating system, or language.

The fourth objective of reengineering is to achieve greater reliability. Although it is possible that the reliability never was very high, more likely, over time and with multiple changes, there have been 'ripple effects' [14], one change causing multiple additional problems. As maintenance and changes continue, the reliability of the software steadily

decreases to the point of unacceptable.

2.2.3 Software Reengineering Concepts

Reengineering is an umbrella term which covers many forms of system improvement, many of which are tool supported [105]. The concepts introduced in software reengineering are based on the software development levels of abstraction shown in Figure 2-1. Each level corresponds to a phase in the development life cycle and defines the software system at a particular level of detail (or abstraction). The conceptual abstraction level is the highest level of abstraction. Here the concept of the system – its reason for existence – is described. At this level, functional characteristics are described only in general terms. In the requirement, abstraction level functional characteristics of a system are described in detailed terms. In these first two levels internal system details are not mentioned. In the design abstraction level system characteristics such as architectural structure, system components, interfaces between components, algorithmic procedure and data structures are described. The implementation abstraction level is the lowest level. Here a system description focuses on implementation characteristics and is represented in a language understood by a computer.

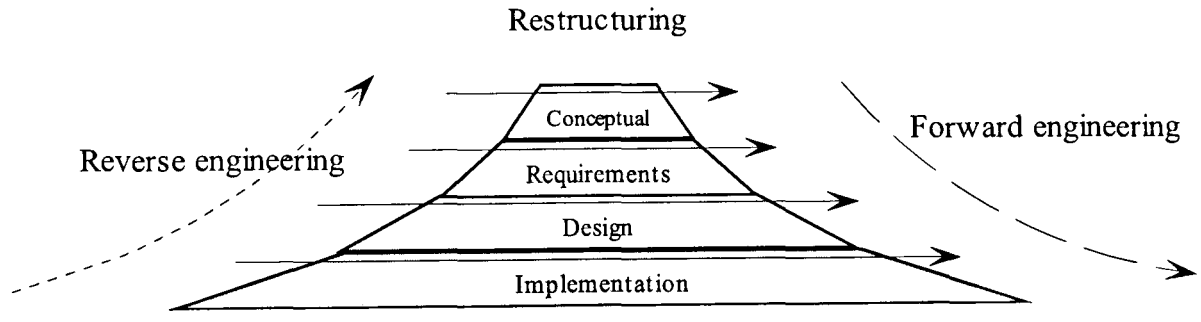

Figure 2-1 Levels of Abstraction

The levels of abstraction present a model for reengineering process crossing these levels. Defined in [13], the process of reengineering computing system involves three main steps: reverse engineering, restructuring and forward engineering.

Reverse engineering is the process of analysing a system in order to obtain and identify major system components and their inter-relationships and behaviours. It involves the extraction of higher-level specifications from the original system. Reverse engineering moves from low-level implementation to high-level abstractions. This involves extracting design artifacts and building or synthesising abstractions that are less implementation dependent.

Restructuring is the process of creating a logically equivalent system from the given one. This process is performed at the same level of abstraction and does not involve semantic understanding of the original system.

Forward engineering is the process of developing a system starting from the requirement specification and moving down towards implementation and deployment. Forward engineering moves from high-level abstractions and logical implementation independent designs to the physical implementation of the system. A sequence from requirements through design to implementation is followed.

Under the spectrum of the reengineering process, there are several activities which are described as follows [125].

- **Abstraction** is a process of generalisation, removing restrictions, eliminating details, removing inessential information [112]. Abstract specification says what a program does without necessarily saying how it does and has more potential implementations.

- **Redocumentation** is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views intended for a human audience.

- **Design Recovery** is a subset of reverse engineering to recreate design abstractions from a combination of code, existing design documentation (if available), personal experience and general knowledge about problem and application domains.

- **Program understanding** implies always that understanding begins with the source code while reverse engineering can start at a binary and executable form of the system or at high-level descriptions of the design. The science of program understanding includes the cognitive science of human mental process in program understanding that can be achieved in an ad hoc manner.

- **Refinement** is an inverse activity of abstraction. It is to refine abstract specification towards to low-level implementation.

- **Re-code** is changes to implementation characteristics. Language translation and control flow restructuring are source code level changes. Other possible changes include conforming to coding standards, improving source code readability, renaming program items, etc.

- **Re-design** is changes to design characteristics. Possible changes including restructuring design architecture, altering a system's data model as incorporated in data structures, or in a database, improvements to an algorithm, etc.

- **Re-specify** is changes to requirement characteristics. This type of change can refer to changing only the form of existing requirements. For example, taking informal requirements expressed in English and generating a formal specification expressed in a formal language. This type of change can also refer to changing system requirements, such as the addition of new

requirements, or the deletion or alteration of existing requirements.

2.3 Program Transformation

The idea that program transformations could be used for software maintenance and evolution by changing a specification and re-synthesising was suggested in the early 80s [10]. Porting software and carrying out changes using transformations were suggested and demonstrated in the late 80s [7]. Simple source-to-source 'evolution' transforms used to change specification code was suggested by Feather [42]. Evolution transforms lead to the key idea of using both correctness-preserving transforms to change nonfunctional properties of software and to use non-correctness preserving transforms to change functional properties of a software system. Theory about how to modify programs incrementally using 'maintenance delta' transformations and previously captured design information was suggested in 1990 [11].

It is very often that a software system has to migrate from one development language to another in order to be able to fulfil the required evolutionary changes [90]. Program transformation has acted as a fundamental technique for evolutionary changes and various software reengineering activities. This section reviews the concepts, the taxonomy, the technical foundations and the transformation systems within the program transformation scope.

2.3.1 Definition of Program Transformation

Any programming language has three components: syntax, semantics and pragmatics [99].

- Syntax defines the formal relations between the constituents of a language, thereby providing a structural description of the various expressions that make

up legal strings in the language. Syntax deals solely with the form and structure of symbols in a language without any consideration give to their meaning.

- Semantics describes the behaviour that a computer follows when executing a program in the language. This behaviour can be disclosed by describing the relationship between the input and output of a program or by a step-by-step explanation of how a program will execute on a real or an abstract machine.

- Pragmatics includes issues such as ease of implementation, efficiency in application and programming methodology.

Program transformation is the act of changing one program into another. It is often important that the derived program be semantically equivalent to the original, relative to a particular formal semantics. The operation is to alter the syntax of the program but preserve the semantics. The languages in which the program is transformed and the resulting program are written are called the source and target language respectively [108].

2.3.2 Taxonomy of Program Transformation

Program transformation has applications in many areas of software engineering such as compilation, optimisation, refactoring, program synthesis, software renovation and reverse engineering. Visser [106] distinguish these applications as two main scenarios, shown as Table 2-1, i.e., the one in which the source and target language are different (translations) and the one in which they are the same (rephrasings).

Translation

In a translation scenario, a program is transformed from a source language into a

program in a different target language. Translation scenarios can be distinguished by their effect on the level of abstraction of a program. Although translations aim at preserving the extensional semantics of a program, it is usually not possible to retain all information across a translation. Translation scenarios can be divided into synthesis, migration, reverse engineering, analysis and so forth.

Translation	**Rephrasing**
• Migration	• Normalisation
• Synthesis	– Simplification
– Refinement	– Desugaring
– Compilation	– Weaving
• Reverse engineering	• Optimisation
– Abstraction	– Specialisation
– Decompilation	– Inlining
– Architecture extraction	– Fusion
– Software visualisation	• Refactoring
• Analysis	– Design improvement
– Control-flow analysis	– Obfuscation
– Data-flow analysis	• Renovation

Table 2-1 A Taxonomy of Program Transformation

Synthesis

Program synthesis is a class of transformations that lower the level of abstraction of a program. In the course of synthesis, design information is traded for increased efficiency. In refinement [100] an implementation is derived from a high-level specification such that the implementation satisfies the specification. Compilation [6, 82] is a form of synthesis in which a program in a high-level language is transformed to machine code. This translation is usually achieved in several phases. Typically, a high-level language is first translated into a target machine independent intermediate representation. Instruction selection then translates the intermediate representation into machine instructions. Other examples of synthesis are parser and pretty-printer generation from context-free grammars [2, 16]

Migration

In migration, a program is transformed to another language at the same level of abstraction. This can be a translation between dialects, for example, transforming a Fortran77 program to an equivalent Fortran90 program or a translation from one language to another, e.g., porting a Pascal program to C.

Reverse Engineering

The purpose of reverse engineering [12, 24] is to extract from a low-level program a high-level program or specification, or at least some higher-level aspects. Reverse engineering raises the level of abstraction and is the dual of synthesis. Examples of reverse engineering are decompilation in which an object program is translated into a high-level program, architecture extraction in which the design of a program is derived, documentation generation and software visualisation in which some aspect of a program is depicted in an abstract way.

Analysis

Program analysis reduces a program to one aspect such as its control flow or data flow. Analysis can thus be considered a transformation to a sublanguage or an aspect language.

Rephrasing

Rephrasings are transformations that transform a program into a different program in the same language, i.e., source and target language are the same. In general, rephrasings try to say the same thing in different words thereby aiming at improving some aspect of the program, which entails that they change the semantics of the program. The main

subscenarios of rephrasing are normalisation, optimisation, refactoring and renovation.

Normalisation

A normalisation reduces a program to a program in a sublanguage, with the purpose of decreasing its syntactic complexity. Simplification is a more general kind of normalisation in which a program is reduced to a normal (standard) form, without necessarily removing simplified constructs. For example, consider transformation to canonical form of intermediate representations and algebraic simplification of expressions. Note that normal form does not necessarily correspond to being a normal form with respect to a set of rewrite rules.

Optimisation

An optimisation [82] is a transformation that improves the run-time and/or space performance of a program. Example optimisations are fusion, inlining, constant propagation, constant folding, common-subexpression elimination and dead code elimination.

Refactoring

The term refactoring was originally introduced by Opdyke in his PhD thesis [86]. Fowler et al [44] defines a refactoring is "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour*". As for the methods used in refactoring, it generally refers to transformations, which move code around or rename identifiers. In practice, refactoring is restricted to restructuring transformations: moving code and renaming code. For example, converting a recursive algorithm to an iterative one would not normally be considered a 'refactoring' operation, nor would replace an algorithm with a completely

different one that produces the same output.

Renovation

In software renovation, the extensional behaviour of a program is changed in order to repair an error or to bring it up to date with respect to changed requirements. Examples are repairing a Y2K bug, or converting a program to deal with the Euro.

2.3.3 Technology Foundations

According to Mens [76], program transformation process consists of a number of distinct activities related to the following questions:

Where should the software be transformed and which transformation(s) should be applied to the identified places?

To identify the parts of software for program transformation normally is applied with the step of proposing transformations together. Kataoka et al. implemented the *Daikon* tool to indicate where transformations might be applicable by automatically detecting program invariants [61]. The main problem with this approach is that it requires dynamic analysis of the runtime behaviour. Nonetheless, the approach is complementary to other approaches that rely on static information. Balzinska et al. use a clone analysis tool to identify duplicated code that suggests candidates for transformations [9]. Ducasse et al. sketch an approach to detect duplicated code in software and propose transformations that can eliminate this duplication. The approach is based on an object-oriented meta model of the source code and a tool that is capable of detecting duplication in code [37]. In FermaT [118], pattern variables are constructed for matching the program schema and identifying the suitable transformations to determine where to be transformed and which transformation to be used.

An important issue about this step is that identification of which transformations to apply can be highly dependent on the particular application domains. The characteristics of the specific domains can result in completely different transformations.

How does the applied transformation preserve behaviour?

By definition, a transformation should not alter the behaviour of the software. Unfortunately, a precise definition of behaviour is rarely provided or may be inefficient to be checked in practice [76]. The original definition of behaviour preservation as proposed originally states that, for the same set of input values, the resulting set of output values should be the same before and after the transformation. In many application domains, requiring the preservation of input-output behaviour is insufficient since many other aspects of the behaviour many be relevant as well. This implies a wider range of definitions of behaviour that may or may not be preserved by a transformation, depending on domain-specific or even user specific concerns.

The preservation property can either be checked statically or dynamically [76]. One needs to remove the restrictions imposed by static conservative approximations by taking more dynamic information into account. However, one should be aware that even then it is impossible to guarantee full behaviour preservation in its generality. Moore and Bennett propose a more dynamic notion of call preservation, where the transformation guarantees that the same messages in a class will be sent in the same order [81]. Ward provides a formal imperative language with formally defined semantics [118]. A fundamental property of the semantics of WSL is that in order to prove a refinement or equivalence relation between two programs, it is sufficient to prove an implication or equivalence between the corresponding weakest preconditions [34, 98].

How is the effect of the transformations on quality characteristics assessed?

For any piece of software, its external quality attributes can be specified such as reusability, performance and so on. Program transformations that alter the program can be applied to improve the quality of software. To achieve this, each transformation has to be analysed according to its particular purpose and effect. Some transformations remove code redundancy, some raise the level of abstraction, some enhance the reusability, etc. This effect can be estimated to a certain extent by expressing the transformations in terms of the internal quality attributes they affect (such as size, complexity, coupling and cohesion).

In the context of logic and functional programs, restructuring transformations typically have the goal of improving program performance while preserving the program semantics [94, 98] . In the context of object-oriented programs, Demeyer [31] concludes that the program performance gets better after the transformation that replaces conditional logic by polymorphism because of the efficient way in which current compiler technology optimises polymorphic methods.

To measure or estimate the impact of a transformation on quality characteristics, many different techniques can be used. Examples include, but are not limited to, software metrics, empirical measurements, controlled experiments and statistical techniques. Kataoka et al. propose coupling metrics as an evaluation method to determine the effect of transformations on the maintainability of the program [127]. Tahvildari and Kontogiannis encode design decisions as soft-goal graphs to guide the application of the transformation process [104]. These soft-goal graphs describe correlations between quality attributes. The association of transformations with a possible effect on soft-goals addresses maintainability enhancements through primitive and composite transformations. Tahvildari and Kontogiannis use a catalogue of object-oriented metrics

as an indicator to automatically detect where a particular transformation can be applied to improve the software quality [103]. This is achieved by analysing the impact of each transformation on these object-oriented metrics.

2.3.4 Transformation Systems

A program transformation system is determined by the choices it makes in program representation and the programming paradigm used for implementing transformations. It leverages an engineer-provided base of 'transforms' to automate analysis, modification and generation of software, enhancing productivity and quality over conventional methods [12]. Transformation system technology has matured to the point where these activities are practical on large scale, production software systems and offer large productivity and quality increments to engineering organisations using them. This section will review a number of existing program transformation systems.

✧ ***DMS Software Reengineering Toolkit***

The Design Maintenance System (DMS) Software Reengineering Toolkit [3, 12] is a set of tools for automating customised source program analysis, modification or translation or generation of software systems, containing arbitrary mixtures of languages ('domains'). The term 'software' for DMS is very broad and covers any formal notation, including programming languages, markup languages, hardware description languages, design notations, data descriptions, etc.

DMS provides generalised compiler technology for automating custom analysis, modification and generation of large software system sources. It includes:

- Unicode-based lexer generator. DMS levers capture binary values of lexemes, all comments and source file positioning information. Support is provided for

INCLUDE files and other preprocessor issues.

- Context-free Generalised Left-to-right Rightmost derivation (GLR) parser generator. Automatic construction of Abstract (not concrete) Syntax Trees (AST). Handling of local and conventional ambiguities. Parse-time semantic checking is possible.

- Pretty printer generator. DMS pretty printers can pretty-print ASTs according to custom pretty printing rules, or 'fidelity' print, preserving as much of the original formatting as possible.

- Multi-pass attribute evaluator generator. Attributes computed from one pass are available in following passes. Attribute evaluation occurs in parallel, based on data dependencies.

- Symbol table support for both conventional and unusual scoping rules.

- Surface-syntax pattern and rewrite rule specification. Patterns/rewrites written in the notation of the target language, or in both source and target notations if different languages. Conditional rewriting, with optional procedural attachment. Optional procedural attachment for Right-hand-side construction. Pattern and rule set compositions.

- Associative/Commutative rewrite engine.

- Scalable foundations Parallel execution based symmetric multiprocessing in PARLANSE, an internal parallel language for symbolic execution. Handles tens of thousands of files comprising several million lines on Windows platforms.

It is claimed by DMS that their toolset predefines a number of legacy languages for use with DMS, such as ANSI C and C++ with a built in preprocessor, COBOL (ANSI 85/IBM VS II) with built in preprocessor, Java 2.0, C#, HTML 4.0, XHTML, Internet Explorer dialect, PHP, ISO Pascal and (Borland) Object Pascal, Ada83/95, Fortran77/90/95, ECMAScript (JavaScript), XML, Verilog, VHDL and so on.

✧ ***TXL***

The tree transformation language TXL [29] is a language designed for specifying rule-based source-to-source transformations, claiming to be a general purpose transformation system. Due to its long history started as early as 1990, this tool has been used in many different industrial projects.

In the transformation rules, a mix of concrete and abstract syntax can be used. The TXL engine applies all transformation rules repeatedly in a nondeterministic way until no more rules succeed. When a rule is applied, it is possible to call other rules that operate on a supplied argument that could for example be a child node. This provides a basic mechanism of custom traversals over the input tree, but it is not very powerful. There is no query support, but template matching support is quite powerful. It is however unclear if and how this functionality allows one to retrieve input from an arbitrary location in the input.

Due to the weak traversal support and due to the bad scaling characteristics of this approach this is not a very good solution. A possibly better approach, which is also aimed at allowing for transformations in which source and target language are different, is to use several intermediate grammars and apply multiple successive rewrites from one grammar to the next.

✧ ***Alchemist***

This approach described in [70] is primarily of interest because it was one of the first program transformation environments and because it is based on TT- grammars [63], a paradigm not commonly used but that looks interesting because it has been intended exactly for the definition of program transformations.

Transformations in Alchemist are based on grammar productions. The tool allows specification of the transformation of a grammar production of the source into the production of the target. Transformations are defined completely in terms of grammar productions, which actually is a mix of concrete and abstract syntax but in the context of this evaluation, it is considered as abstract syntax. After specifying the exact transformation, the system will perform a simple traversal over the input to transform every node. This traversal is implicit and cannot be customised in any way. Furthermore, it is not possible to add user-defined traversals over the input. Support for data acquisition and queries is absent, which is recognised by the developers.

Although tree transformation grammars can be very good at 1-to-1 and some local-to-local forward transformations, there is absolutely no support for traversals, queries or data acquisition. This makes all other transformation scopes very hard to implement. It is unclear what a reverse transformation should look like in the context of tree transformation grammars. It is therefore no surprise that Alchemist does not support this. Multi-stage transformations are possible in theory, but no references have been found in the literature on this functionality.

✧ ***ASF+SDF***

The ASF (Algebraic Specification Formalism) +SDF (Syntax Definition Formalism) meta-environment [33] provides a complete environment for program transformations

based on the term rewriting paradigm. One of its qualities is that it provides strong type checking on transformation rules. Another reason that makes it an interesting tool to study is because it has been used in a number of industrial projects. The equations that define the transformation are written in a concrete syntax.

This concrete syntax is fully parsed and the resulting AST is used internally. Besides the equations, a transformation specification also consists of a complete grammar definition of both input and output, which makes it possible to generate any type of output easily.

Traversals are fairly well developed. There are two basic traversals, bottom up and top down. It is also possible to define new traversals by manually specifying inside every rule how a transformation should recurse into its children. Due to the tight coupling of the transformation and this type of traversals, it is unfortunately not possible to separate this functionality, making reuse of custom traversals very difficult.

There is no special query functionality that is allowed for easy lookups of information. Although the rich equation functionality can be used for expressing certain queries, it does not provide a very compact syntax and is only practical for querying the current sub tree not allowing for easy access to other parts of the input tree.

There are a number of special features that are allowed for easy data acquisition. First of all it is possible to collect data in extra attributes during a manual traversal. This suffers from the problem that the addition of a single attribute will affect many equations, which presents a scaling problem. The usage of the predefined traversal types accumulator or transformer provides a much nicer approach. For these predefined traversal types the system generates default rules that will automatically handle the traversal over terms for which no traversal is specified. It even takes extra arguments into account, making data acquisition easy. This however only works nicely if the traversal can be captured in a

basic bottom up or top down traversal. If this is not the case there is the option of overriding the traversal at some key points, but these overrides should specify exactly all extra arguments used, introducing again the same scaling problem as in the manual traversal.

✧ ***XML/XSLT***

The very popular functional transformation language XSLT (Extensible Stylesheet Language Transformation) [109] can be used to transform Extensible Markup Language (XML)documents into arbitrary output and cannot be overlooked when studying program transformations, although up to now it has not been used much for program transformation.

A typical XSLT rewrite rule, or template in XSLT terms, receives as input an XML document, for example, the abstract syntax for a Java file. The right hand side of these templates is the output also in terms of XML. However, since simple string values can be used as XML values, concrete syntax can be used just as well. Although this approach is widely used, it does not provide any format checking and suffers from escaping problems. Another reason is that XSLT was not designed to be used like this, which is supported by for example the new XSLT 2.0 specification that allows for applying a transformation to the result of another transformation. This is of course only possible if all transformation results are specified in terms of XML. Although only XML should be output, this XML can of course be abstract syntax for any language and can be pretty printed to the concrete syntax of that language. It is possible to define traversals by manually recursing into the children of a node, but this does not allow for a clean separation of traversal and rewriting.

For this reason, intelligent traversals are hard to write and maintain. This is compensated

by the powerful XPath matching functionality [110], which can be used for defining the nodes a template should be applied on. It allows not only for matching against a specific node, but also against the context a node occurs in, for example by putting restrictions on it's ancestors. Query support, through XPath expressions, is very well developed. These expressions allow selection of nodes precisely, using a compact and easy to learn syntax. Furthermore, it always allows direct access to the whole tree and not only to just the current sub-tree. Data acquisition is easy, primarily because of the powerful queries. Furthermore, it is also possible to manually pass extra arguments to recursive template applications allowing propagation of data through a traversal. Finally, there are variables inside templates and global variables that can hold arbitrary values, making sharing of data very easy.

✧ ***Stratego***

The special purpose transformation language Stratego is based on term rewriting [107] and can be seen as the first incarnation of the strategic programming paradigm. It provides numerous features such as hygienic variables, concrete object syntax and dynamic rules that make it very suitable for implementing program transformations. Internally all transformations operate on an AST. The programmer has the choice between abstract syntax, usually preferable for small transformations or small code fragments that are ambiguous in concrete syntax or abstract syntax, that can be mixed into a Stratego program using a fully customisable syntax for defining quotation and antiquotation [18]. The output of a transformation always is an AST that can be pretty-printed to concrete syntax. Since the programmer is completely free to choose the format of the output AST any type of output can be generated.

Because of the strategic programming paradigm used in Stratego traversal support is excellent. Generic term traversal is readily available through the usage of congruence

operators and generic basic traversal strategies such as 'all' and 'one'. These can be combined using strategy combinators for sequential composition, non-deterministic choice and many others, to build very complex traversals. In fact, Stratego includes a library with many common generic traversal strategies such as top down, bottom up, collect, innermost and many others, which are evidence of the power of this mechanism.

There is no standard query functionality available, but powerful pattern matching support in combination with intelligent traversals provides access to data. Although this approach at first sight only seems to give access to data in the sub tree of the input of a transformation, scoped dynamic rewrite rules [17] can be used to access the initial input tree at any time during the transformation.

Furthermore, a study [121] shows the user-defined extension with XPath-like [109] queries using a custom, application-specific, syntax. Data acquisition is possible in a number of ways; scoped dynamic rewrite rules provide a very interesting approach of making data available elsewhere in a way nicely fitting in the term rewriting paradigm. Furthermore, it is possible to tuple the normal input tree with other data and carries along this information during the traversal. This can present a scaling problem, since addition of new arguments will likely force modification of many rules. An approach that scales much better is the definition of a special traversal that holds an environment whose values can be used inside every transformation. The standard Stratego library already contains a number of these traversal strategies such as env-topdown and env-bottomup.

✧ ***Tom***

Tom [49, 80] is a software environment for defining transformations in Java. It is particularly well suited for programming various transformations on trees/terms and XML based documents. Since Tom is an extension of C or Java, it is naturally a general

purpose language that can be used to implement a large class of applications. The main contribution of Tom is to add pattern matching facilities to C and Java. The application domain of Tom becomes naturally related to the manipulation of structured tree/term based objects. By introducing high-level, typed and structured constructs, Tom brings C or Java closer to algebraic and functional style programming languages.

✧ ***FermaT Transformation Engine***

The FermaT transformation engine [112-119, 125] is the latest version which has evolved over two decades from a laboratory tool to a practical system based on the transformation-intensive language WSL and its transformation theory. The details of this tool and its theory are given in Section 3.6. The proposed research is implemented based on the FermaT transformation engine. The choice of FermaT as an experiment platform relies on the following reasons against to the other transformation systems.

- The flexibility and extendibility of WSL and *Meta*WSL. Using WSL as an intermediate language is beneficial to the application of transformations on various program languages. It is not required to develop different transformations for different languages. *Meta*WSL as an extension of WSL is for manipulating programs particularly and is an ideal means to develop transformations. The layered structure of WSL facilitates the language extension based on the existing constructs. That makes WSL adaptive for more cases than the other systems, which are developed for a particular language only.

- The reengineering-oriented transformations. In addition to the transformations used to operate the same abstraction levels of software, FermaT transformation engine has the transformations for abstraction and refinement

which are crucial for reverse engineering and forward engineering. Its transformations can facilitate the reengineering process.

- The rich transformation bank. By the academic research and the practical industrial applications for the two decades, there have been a large number of transformations developed. The transformations were proven theoretically before their development. Therefore, FermaT transformation engine is suitable to experiment the research on transformation prediction.

2.4 Software Goal-Driven Requirement Specification

The proposed approach, the target driven program transformation steps predication (TDPTSP) towards reengineering targets, involves a transition from requirements to software entities or to a representative model of the implemented system. The input is one target or multiple targets, the correlated metrics and the existing system. The output expected is, for the target(s), a set of transformation candidates and their execution sequence. The output is used as knowledge to assist the transformation process.

The target referred here can be regarded as a kind of software requirement informally because it is proposed according to the needs of software reengineering. This section will introduce the relevant concepts and techniques used in software requirement specification which can be used for modelling and specifying targets.

2.4.1 Requirement Concepts

It is common to make a distinction between *user* and *system requirements* [64], which are defined as follows.

- User requirements (or stakeholder requirements) define what the user requires

of the software or system as a whole. User requirements are informal. They are written by the user or taken down by a system analyst in consultation with the user.

- System requirements (requirement specifications) define what is required of the system as a whole, either hardware or software. Analysts develop system requirements, as a refinement of user requirements, by translating them into engineering terms.

Requirement specification commonly identifies four major classes of requirements [15]:

- Functional requirements which specify a function that a system or a system component (i.e. software) must be capable of performing;

- Non-functional requirements which state the characteristics of the system to be achieved that are not related to its functionality, i.e. its performance, reliability, security, maintainability, availability, accuracy, error-handling, capacity, types of users, changes to be accommodated, level of training support, etc.;

- Inverse requirements which describe constraints on the system expressed in terms of what the system will not be able to do, e.g. in relation to software safety or security requirements; and

- Design and implementation constraints which state the boundary conditions on how the required software is to be constructed and implemented.

As the quality of requirements has a pivotal role in reaching the quality of the final software product, it is important that the requirements specification meets some well-recognised quality standard. Therefore, it should identify only the information that

is necessary and actually useable in the development of the software project. All such information should be expressed in unambiguous and consistent terms and be complete and verifiable. Individual requirements should be prioritised to allow scheduling of all development tasks and should the user requirements change; the specification must be easily modifiable. Any software development products must also be traceable back to the original requirements statements.

2.4.2 Goal Targeting Modelling

Within requirement engineering, the notion of goal has increasingly been used. Goals generally describe targets which a system should achieve through cooperation of actors in the intended software and in the environment [71]. Goals are central in some requirement engineering frameworks and can play a supporting role in others.

The term *soft goal* is used in connection with modelling languages and especially with goal-oriented modelling. Soft goals can represent [25]:

- Non-functional requirements

- Relations between non-functional requirements

Normally a goal is a very strict and clear logical criterion. It is satisfied when all sub-goals are satisfied. However, in the non-functional requirements you often need more loosely defined criteria, like satisficeable or unsatisficeable. Soft goals are goals that do not have a clear-cut criterion for their satisfaction: they are satisfied when there is a sufficient positive and little negative evidence for this claim, while they are unsatisficeable in the opposite case.

Goals can be related to other soft-goals in terms of relations such as AND, OR, + or –

[25, 83, 84]. The meaning of these relations has been shown as Table 2-2. Goal-oriented analysis amounts on an intertwined execution of the three types of analysis sketched here, namely analysis of non-functional requirements as soft-goals, of functional requirements as goals and conflict analysis. The analysis can be declared complete when all relevant goals (soft or otherwise) have been operationalised in terms of constraints on and functions to be performed by, the new system. Goal-oriented analysis focuses on the description and evaluation of alternative and their relationship to the organisational targets behind a software development project.

$AND\,(G, G_1, G_2, \cdots, G_n)$	Goal G is satisficed when all of $G_1, G_2, \cdots, G_n$ are satisficed and there is no negative evidence against it. Goal G is unsatisficed and there is one of $G_1, G_2, \cdots, G_n$ is unsatisficed and there is no positive evidence for it.
OR $(G, G_1, G_2, \cdots, G_n)$	Goal G is satisficed when one of $G_1, G_2, \cdots, G_n$ is satisficed and there is no negative evidence against it. Goal G is unsatisficed if all of $G_1, G_2, \cdots, G_n$ are unsatisficed and there is no positive evidence for it.
$+\,(G_1, G_2)$	Goal G_1 contributes positively to the fulfilment of goal G_2.
$-\,(G_1, G_2)$	Goal G_1 contributes negatively to the fulfilment of goal G_2.

Table 2-2 Goal Relationship

2.5 Software Metrics

Software metrics as a subject area has been over 30 years old since Maurice Halstead published his first paper [50], which was the beginning of the first long-term software metrics research effort. Since then software metrics have become a significant part of software engineering.

Metrics are critical to any engineering discipline and software engineering is no

exception. Software metrics can be used throughout the software life cycle to assist in cost estimation, quality control, productivity assessment and project control and can be used to help asses the quality of technical work products and to assist in tactical decision making as a project proceeds.

In the definitions of software metrics, three terms, 'measure', 'measurement' and 'metrics', must be noticed and distinguished, because definitions of these terms can easily become confusing. Within the software engineering context, these terms are defined as follows:

- A *measure* provides a quantitative indication of the extent, amount, dimensions, capacity, or size of some attributes of a product or process [92].

- *Measurement* is the act of determining a measure [92]. It is the process of empirically and targetly assigning numerical results to the attributes of software in such a way as to describe the software.

- A *software metric* is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.

Other important definitions are listed as follows following classical texts in the science of measurement [65, 66, 95].

- An attribute is a feature or property of an entity.

- Direct measurement of an attribute is measurement that does not depend on the measurement of any other attribute.

- Indirect measurement of an attribute is measurement that involves the measurement of one or more other attributes. (It is often useful in making

visible the interactions between direct measurements.)

After three decades, plenty of metrics have been developed for different measurements. Those metrics are used to serve the assessment of the properties of software [43, 88].

2.6 Automation of Program Transformation for Software Reengineering

A core technique of the proposed approach is 'prediction'. The prediction in the approach refers to find the applicable transformations and the execution sequence of them, which contribute the specified targets. The process of the prediction is actually to search a set of suitable solutions to improve automation of program transformation towards reengineering targets. With regard to the aspect of the proposed research, a few research works related to transformation automation are reviewed and compared in this section for the further discussion on the proposed approach.

Tracing back to 1970's, target-driven program transformation has drawn attention. Wegbreit [120] proposed that program transformation can be made goal-directed. In their approach, the execution performance goals are established to direct the process of program transformation which is carried out by local simplification, partial evaluation of recursive functions, abstraction of new recursive function definitions from recurring subgoals and generalisation of expressions as required obtaining compatible subgoals. The goal is represented as expression. The closed-form algebraic expressions which describe execution behaviour are derived to specify the program's computation cost and program's output characteristics. Their transformations are used to remove the program segments whose computation costs are not accounted for in the estimates of minimum cost. The approach did not realise the automation of the transformation but suggested the idea of goal-driven which can direct the implementation of transformations.

The action to improve the automation of program transformation by prediction approach can be regarded as a search problem in that it is needed to find the candidates among a large number of transformations and determined their execution sequence. The proposed problem has a tight link with Search Based Software Engineering (SBSE). SBSE is a fast growing field in that search based solutions have a track record of success in the domain of software engineering, characterised by a large number of potential solutions, where there are many complex, competing and conflicting constraints and where construction of a perfect solution is either impossible or impractical [126]. Growth in interest is fueled by the way in which search techniques can be applied right across the life-cycle of software and the speed with which the techniques can be mastered and deployed to produce results [53]. These techniques provide robust, cost-effective and high quality solution for several problems in software engineering. It is precisely these factors which make robust meta-heuristic search-based optimisation techniques readily applicable [52]. Meta-heuristic algorithms, such as genetic algorithms [48], simulated annealing [67], A* search [102] and tabu search [46] have been applied successfully to a number of engineering problems.

In order to reformulate software engineering as a search problem, it is necessary to define [52]:

- The representation of a candidate solution. This is critical to shaping the nature of the search problem.

- A fitness function (defined in terms of this representation) which is the characterisation of what is considered to be a good solution. When constructing the fitness function, in most case, metrics to measure software properties are used as the fitness function [51].

- A set of manipulation operators. Different search techniques use different operators. As a minimum requirement, it will be necessary to mutate an individual representation of a candidate solution to produce a representation of a different candidate solution.

Recently, quite a few researches have been devoted in the area of SBSE, to search the optimal solutions, including the one of program transformation in software reengineering.

As a pioneer work in the subject of SBSE for program transformation, Fatiregun et al. proposed to using genetic algorithms for evolving transformation sequences [41]. In this work, they reformulated the transformation problem as a search issue for optimisation, provided evidence that evolutionary and/or local search can be used to evolve good transformation sequences and investigate the difference between Hill-Climbing (HC) and the Genetic Algorithms (GA). A system is proposed so that transformation sequences for a variety of target functions can be dynamically generated. As an experiment, the number of Lines of Code (LOC) is used as the fitness function to compare the effect of the transformation sequence generation by different search algorithms. In their transformation sequence, the transformations that could transform an entire program in one single step are not included. In addition to the atomic ones that work on pairs of nodes on abstract syntax tree, the operations which move the current cursor position on the abstraction syntax tree of the program are also included. There are two points to argue w.r.t their research. (1) The transformations that could transform an entire program in one single step should be included in the approach. The selection of the transformations should be complete. (2) The generation of the transformation sequences through GA or HC does not take the expertise and domain knowledge into account. Expertise is also a crucial knowledge and should not be neglected. The efficiency and the correctness of the result could be doubted due to the two factors.

Tahvildari and Kontogiannis [133, 134] proposed a quality driven object-oriented reengineering transformation framework that allows for quality requirements of the target system to be modelled as soft-goals and transformations to be applied selectively towards achieving specific quality requirements for the target system. These transformations can be applied as a series of the iterative and incremental steps to the source code. The authors used a revised A* heuristic algorithm to determine the transformation solutions. An evaluation procedure can be used at each transformation step to determine whether specific goals have been achieved. However, the transformations used in their approach were not proved semantic preserving. The transformation result cannot ensure the behaviour of the legacy system unchanged while its qualities are improved as desired. The evaluation function defined for implementing the A* search is not precise in that the cost function is the combination of the metrics which can be valued in very different ranges. This could cause the neglect of the transformation impact on some metrics and result in an inaccurate cost evaluation. In addition, the heuristic function defined in their approach is so imprecise that the A* search could fail to find the optimised solutions as the authors claimed.

Zou et al. [103, 104] presented a software migration process to transform a subject system from its original procedural language implementation to an object oriented design without altering its external behaviour. In this context, the migration process is denoted by a sequence of transformations each one of which alters the state of the system being migrated. In order to identify the optimal sequence of transformations that can be applied at any given state of the migration process, the authors used a state transition system modelled as Markov type chains and the Viterbi algorithm for the optimisation. In the transition model, every transformation is denoted with a quality factor that indicates the impact on specific code features when applying each transformation in a migration. However, their approach assumes that the final state is well defined, for

example, such as a migrated object-oriented system which has the desired qualities. Unfortunately, this limitation prevents the application of the framework for continuous system improvement. To apply the framework in more general transformation processes, it must be enhanced with better quality prediction capabilities.

Ryan [97] worked on using search techniques to automate parallelisation for supercomputers and described the application of Genetic Programming (GP) to a real world application area – software reengineering in general and automatic parallelisation specifically. Unlike most uses of Genetic Programming, this book evolves sequences of provable transformations rather than actual programs. It demonstrates that the benefits of this approach are twofold: first, the time required for evaluating a population is drastically reduced and second, the transformations can subsequently be used to prove that the new program is functionally equivalent to the original. The work shows that there are applications where it is more practical to use GP to assist with software engineering rather than to replace it entirely. It also demonstrates how the author isolated aspects of a problem that were particularly suited to GP and used traditional software engineering techniques in those areas for which they were adequate.

Agosta et al [1] proposes a methodology for the co-exploration of the design space composed of architectural parameters and source program transformations. They presented a heuristic technique and used it to efficiently span the multi-target co-design space composed of the product of the parameters related to the selected program transformations and the configurable architecture. In the transformation space of their work, the most important optimisation transformations called passes are classified in three categories: dataflow passes, passes that simplify the control and enlarge the code and passes that modify the access pattern to data. A transformation point in their work is any point in a program where a transformation can be applied.

Cooper et al. [28] focus on searching for sequences of compiler optimisation transforms which work largely on compiled code using biased random sampling. They compare the results of their experiments with those obtained against a fixed set of optimisations in a predetermined order.

2.7 Language for Program Transformation – Wide Spectrum Language (WSL)

2.7.1 Background of WSL

Wide Spectrum Language (WSL) [72, 112-115, 117, 118, 125] has been developed for almost two decades and has been used to build a general approach and a tool [114, 115, 118] for addressing reengineering research issues such as program comprehension and reverse engineering using program transformation and abstraction techniques. The wide spectrum language is so termed because it embraces the whole spectrum from mathematical specifications to executable implementations [125]. WSL contains both specification constructs, such as the general assignment statement and programming constructs, such as while-do loops [129].

Because WSL represents specifications and executable implementations, it is ideal for reengineering and re-documentation purposes. By translating a legacy system's source code to WSL as an intermediate representation, a reengineering tool can derive both the specifications inherent within the WSL representation in order to generate the documentation of the system [125]. The use of WSL as an intermediate language or representation has many advantages including the ability to use standardised transformations and mappings from the intermediate to the target domain and, thus, avoid the 'impedance mismatching' problem between the source and target domain. This allows the reengineering effort to be divided up into smaller steps rather than a

monolithic source to target domain reengineering effort [79]. WSL was developed with several advantages in mind:

(1) WSL has simple, regular and formally defined semantics.

(2) Its syntax is simple, clear and unambiguous.

(3) WSL has the ability to express general specifications in terms of mathematical logic with suitable notation.

(4) WSL is supported by well-developed library of proven transformations which do not require the user to fulfil complex proof obligations before these transformations can be applied.

(5) WSL supplies a suite of constants, functions and procedures which are especially designed to handle lists which represent WSL programs. This collection of procedures and functions is known as *Meta*WSL which is extended from WSL for writing program transformations.

(6) Techniques based on WSL can bridge the 'abstraction gap' between specifications and implementation.

(7) WSL has the ability to scale to large programs and has broad real industry-intensive applications in the world wide range.

(8) WSL has existing tool support as well. The FermaT tool [114, 115, 118] was designed to use WSL and has applications in the following areas:

- Improving the maintainability of existing mission-critical software.

- Translating programs to modern programming languages. FermaT often translates programs written in obsolete assembler language to more modern languages such as C.

- Extracting reusable components from the current system, deriving their specifications and storing the specifications, implementation and development strategy.

- Reverse engineering existing systems to high-level specifications, followed by subsequent reengineering and evolutionary development.

WSL was adopted because, among many reasons, its ability for proven transformations and the ability to represent high and low levels of abstraction which make it well suited for reengineering purposes. Particularly, its possession of a rich transformation bank is ideal to explore the transformation prediction approach.

2.7.2 Intermediate Language for Transformation

2.7.2.1 The Kernel Language

The WSL language is built up in a series of layers from a small a mathematically tractable 'kernel language'. This kernel language is based on infinitary first order logic, which is an extension of ordinary first order logic which allows conjunction and disjunction over (countably) infinite lists of formulae and quantification over finite lists of variables. Expressions and conditions (formulae) in WSL are taken directly from infinitary first order logic. Statements in the kernel language are constructed by combining infinitary logic formulae, lists of variables and statement variables. Four primitive statements and three compound statements are needed to define the whole kernel language. Let P and Q be any infinitary logical formulae and x and y be any finite,

non-empty lists of variables. The primitive statements are [117, 125]:

Add variables: **add** (x) adds the variables in x to the state space and assigns arbitrary values to them. If the variables are already in the state space, then they still get assigned arbitrary values.

Remove variables: **remove** (y) removes the variables in y from the state space if they are present: i.e. it ensures that the variables are no longer in the state space.

Guard: [P] is a guard statement. It always terminates and enforces P to be true at this point in the program without changing the values of any variables. It has the effect of restricting previous non-determinism to those cases which will cause P to be true at this point. If this cannot be ensured then the set of possible final states is empty and therefore all the final states will satisfice any desired condition (including P);

Assertion: $\{Q\}$ is an assertion statement which acts as a partial skip statement. If the formula P is true then the statement terminates immediately without changing any variables, otherwise it does not terminate.

The compound statements are as follows; for any kernel language statements **S1** and **S2**, the following are also kernel language statements:

Sequence: ($S_1 ; S_2$) executes S_1 followed by S_2 ;

Nondeterministic choice: ($S_1 \prod S_2$) chooses one of S_1 or S_2 for execution, the choice being made nondeterministically;

Recursion: ($\mu X.S_1$) where X is a statement variable (taken from a suitable set of

symbols). The statement S_1 may contain occurrences of X as one or more of its component statements. These represent recursive calls to the procedure whose body is S_1.

The kernel primitives have been described as 'the quarks of programming' – rather mysterious objects which cannot be found in isolation (the guard statement cannot be implemented) but which combine to form more familiar objects: combinations which, until recently, were thought to be 'atomic' and indivisible.

2.7.2.2 Extending the Kernel Language

The kernel is a very simple and mathematically tractable language which contains all the operations needed for a programming and specification language. It is relatively easy to prove the correctness of transformations in the kernel language, but the language is not very expressive for programming. The language is extended into an expressive programming language by defining new constructs in terms of the kernel. This extension is carried out in a series of layers, illustrated in Figure 2-2, with each layer building on the previous language level. A series of new language levels is built up, with the language at each level being defined in terms of the previous level; the kernel language is the level zero language which forms the foundation for all the others. Each new language level automatically inherits the transformations proved at the previous level; these form the basis of a new transformation catalogue. Transformation of each new language construction is proved by appealing to the definitional transformation of the construct and carrying out the actual manipulation in the previous level language. This technique has proved extremely powerful and has led to the development of a practical transformation system (FermaT) which implements a large number of transformations. Over the last eighteen years, the WSL language and transformation theory have been

developed in parallel: after a sufficiently complete set of transformations for dealing with that a construct, it is only needed to add the new construct to the language. This is one of the reasons for the success of WSL, as witness by the practical utility of the program transformation tool.

Because of the advantage of WSL, it is possible to extend WSL with new constructs so that the application of the program transformation theory based on WSL can be extended into new domains with new features.

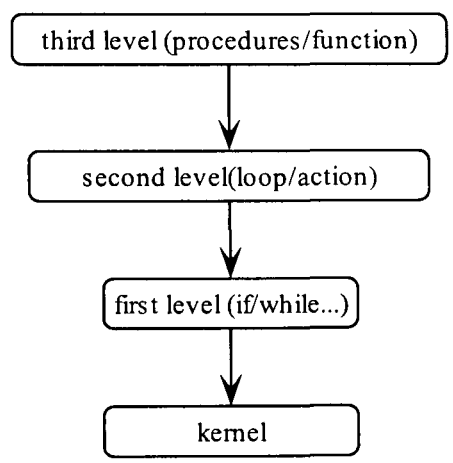

Figure 2-2 WSL Language Levels

The first level language consists of the following constructs:

Sequential composition: The sequencing operator is associative so the brackets can be eliminated:

$$S_1;S_2;S_3;\cdots;S_n =_{DF} (\cdots((S_1;S_2);S_3);\cdots;S_n)$$

***Deterministic choice*:** guards can be used to turn a nondeterministic choice into a deterministic choice:

$$\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi } =_{DF} (([B];S_1)\prod([\neg B];S_2))$$

Specification statement:

$$x := x'.Q \ =_{DF} \ \{\exists x'.Q\};\ \textbf{add}(x');\ [Q];\ \textbf{add}(x);\ [x = x'];\ \textbf{remove}(x')$$

Simple assignment: if Q is of the form $x' = t$ where t is a list of terms that do not contain x' then the assignment can be abbreviated as follows:

$$x := t =_{DF} x := x'.(x' = t)$$

If x contains a single variable, $x := t$ is written as $\langle x \rangle := \langle t \rangle$;

Nondeterministic choice: The 'guarded command' of Dijkstra:

$$\begin{aligned} &\textbf{if } B_1 \to S_1 && =_{DF} (\{B_1 \vee B_2 \vee \cdots \vee B_n\}; \\ &\square\ B_2 \to S_2 && (\cdots(([B_1]; S_1) \textstyle\prod \\ &\quad \cdots && ([B_2]; S_2) \textstyle\prod \\ &\square\ B_n \to S_n\ \textbf{fi} && \cdots)) \end{aligned}$$

Deterministic iteration: a while loop is defined using a new recursive procedure X that does not occur free in S:

$$\textbf{while } B \textbf{ do } S \textbf{ od } =_{DF} (\mu X.(([B]; S) \textstyle\prod [\neg B]))$$

Nondeterministic iteration:

$$\begin{aligned} &\textbf{do } B_1 \to S_1 && =_{DF} \textbf{ while } && (\{B_1 \vee B_2 \vee \cdots \vee B_n\} \textbf{ do} \\ &\square\ B_2 \to S_2 && && \textbf{if } B_1 \to S_1 \\ &\quad \cdots && && \quad \square\ B_2 \to S_2 \\ &\square\ B_n \to S_n\ \textbf{od} && && \quad \cdots \\ & && && \quad \square\ B_n \to S_n\ \textbf{fi od} \end{aligned}$$

Initialised local variables:

$$\textbf{begin } x := t : S \textbf{ end } =_{DF} (\textbf{add}(x); ([x = t]; (S; \textbf{remove } (x))))$$

Counted iteration: Here, the loop body S must not change i, b, f or s:

$$\begin{aligned} \textbf{for } i := b \textbf{ to } f \textbf{ step } s \textbf{ do } S \textbf{ od } =_{DF} & \textbf{ begin } i := b: \\ & \textbf{while } i \le f \textbf{ do } S;\ i := i + S \textbf{ od end} \end{aligned}$$

Block with procedure calls:

$$\textbf{begin } S \textbf{ where proc } X \equiv S' \textbf{.end } =_{DF} S[(\mu X.S')/X]$$

One aim for the design of the first level language is that it should be easy to determine which statements are null potentially.

The level two languages introduce multi-exit loops and Action Systems. Level three adds local variables and parameters to procedures, functions and expression with side effects.

The extendibility of WSL by adding or adjusting language construct levels endues itself with the flexibility when WSL is used for the program written in the other imperative language such as object-oriented programming.

2.7.2.3 The Specification Statement

A simple combination of kernel statements is used to construct the specification statement $x := x'.Q$ where x is a sequence of variables and x' the corresponding sequence of 'primed variables' and Q is any formula. This assigns new values to the variables in x so that the formula Q is true where (within Q) x represents the old values and x' represents the new values. If there are no new values for x which satisfices Q then the statement aborts.

The formal definition of $x := x'.Q$ is:

$$\{\exists x'.Q\}; \textbf{add}(x'); [Q]; \textbf{add}(x); [x = x']; \textbf{remove}(x')$$

The first assertion ensures that the statement aborts if there are no values for the x' variables which satisfy Q. The next two statements add x' to the state space with arbitrary values and then restrict the values to satisfy Q. The final three statements copy

the values from x' to x and then remove x' from the state space. It is assumed that the 'primed variables' are a separate set of variables which are not used outside specification statements.

Any WSL program can be transformed into a single equivalent specification statement. This shows that the specification statement is sufficiently general to define the specification of any program.

By having expressive program constructs in the implementation level and specification statements in the abstraction level, WSL is de facto wide spectrum covering high level and low level of a program.

2.7.2.4 Semantics of the Kernel Language

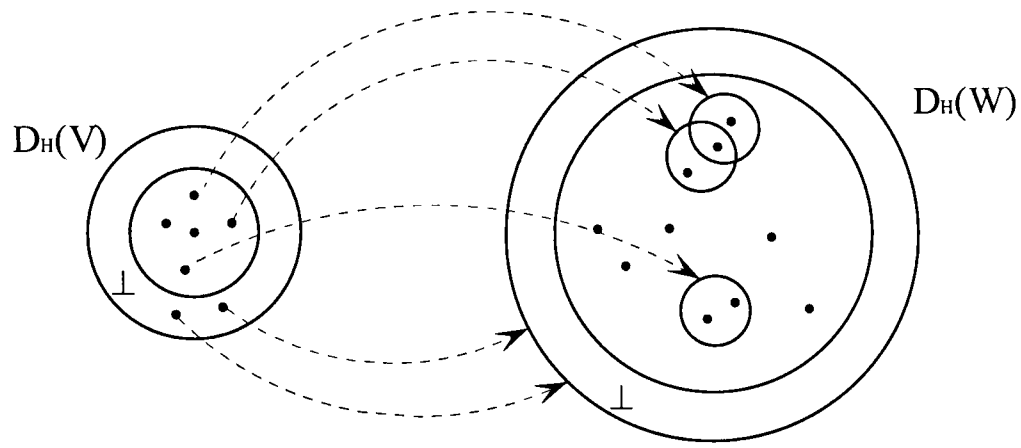

Figure 2-3 State Transformation Illustration

Let V and W be finite sets of variables and H be a set of values.

A state is either the special $\perp$ which indicates nontermination or error, or is function from V to H. This function gives a value (taken from H) to each variable in the state space. The set of all state on V is denoted $D_H(V)$ where $D_H(V) =_{DF} \{\perp\} \cup H^V$, H^V is the defined states on V.

A state predicate is a set of proper states (i.e. states other than $\perp$), with the set of all state predicates denoted $E_H(V)$.

A state transformation is a function which maps a state in V_H to a set of states in W_H, where $\perp$ maps to W_H and if $\perp$ is the output, then so is every other state. The state transformation is illustrated in Figure 2-3. The set of all state transformation from V to W may therefore be defined as:

$$F_H(V,W) =_{DF} \{f : D_H(V) \rightarrow (E_H(W) \cup (D_H(W))) \mid f(\perp) = D_H(W)\}$$

The non-recursive kernel language statements are defined as state transformations as follows:

$$\{e\}(s) =_{DF} \begin{cases} \{s\} & if \quad s \in e \\ D_H(W) & otherwise \end{cases}$$

$$[e](s) =_{DF} \begin{cases} \{s\} & if \quad s \in e \\ \phi & otherwise \end{cases}$$

$$\textbf{add}(s) =_{DF} \{s' \in D_H(W) \mid \forall y \in W.(y \notin x \Rightarrow s'(y) = s(y))\}$$

$$\textbf{remove}(s) =_{DF} \{s' \in D_H(W) \mid \forall y \in W.(s'(y) = s(y))\}$$

$$(f_1; f_2)(s) =_{DF} \bigcup \{f_2(s') \mid s' \in f_1(s)\}$$

$$(f_1 \prod f_2)(s) =_{DF} f_1(s) \cup f_2(s)$$

Three fundamental state transformations in $F_H(V,V)$ are: Ω, Θ, Λ. These give the semantics of the statements **abort**, **null** and **skip**, where **abort** is defined as {false}, **null** is defined as [false] and **skip** is defined as {true}. For each proper $s \in D_H$:

$$\Omega(s) =_{DF} D_H(V) \qquad \Theta(s) =_{DF} \phi \qquad \Lambda(s) =_{DF} \{s\}$$

Recursion is defined in terms of a function on state transformations:

Theorem 2-1 ***Recursion***: Suppose a function F which maps the set of state transformations $F_H(V,V)$ to itself. A recursive state transformation from F as the limit of the sequence of state transformations $F(\Omega), F(F(\Omega)), F(F(F(\Omega))), \cdots$. With the definition of state transformation given above, this limit $(\mu.F)$ has a particularly simple and elegant definition:

$$(\mu.F) =_{DF} \coprod_{n<\omega} F^n(\Omega)$$

where $\coprod$ on a set of state transformation is defined by pointwise intersection:

$$(\coprod X)(s) =_{DF} \bigcap \{f(s) \mid f \in X\}$$

$F^n(\Omega)$ is the 'nth truncation' of $(\mu.F)$: as n increases the truncations get closer to $(\mu.F)$. The later truncations provide more information about $(\mu.F)$ – more initial states for which it terminates and a restricted set of final states. The $\coprod$ operation collects together all this information to form $(\mu.F)$.

With this definition, $(\mu.F)$ is well defined for every function $F: F_H(V,V) \rightarrow F_H(V,V)$.

However if the recursive statement state transformations needs to satisfy the property $F((\mu.F)) = (\mu.F)$ (in other words, to be a fixed point of the F function) then the further restrictions on F is required.

The definition of recursion will be referred for the extension of WSL with object-oriented features. The fixed point approach will be used to define the semantics of inheritance and self-reference relationship.

2.7.3 Program Transformation Theory

2.7.3.1 Refinement and Equivalence

A state transformation can be thought of as either a specification of a program, or as a (partial) description of the behaviour of a program. If f is a specification, then for each initial state s, f(s) is the allowed set of final states. If $\perp \in f(s)$ then the specification does not restrict the program in any way for initial state s, since every other state is also in f(s).

Similarly, if f is a program description, then $\perp \notin f(s)$ means that the program is guaranteed to terminate in some state in $f(s)$ when started in state s.

Program f satisfies specification g precisely when $\forall s.(f(s) \subseteq g(s))$.

A program f_2 is a refinement of program f_1 if f_2 satisfies every specification satisfied by f_1, i.e. $\forall g.(\forall s.(f_1(s) \subseteq g(s)) \Rightarrow \forall s.(f_2(s) \subseteq g(s)))$. It is easy to see that refinement and satisfaction, as defined above, are identical relations.

Theorem 2-2 ***Refinement*****:** Given two state transformations f_1 and f_2 in $F_H(V,W)$, f_2 refines f_1, or f_1 is refined by f_2, written as $f_1 \leq f_2$ if and only if f_2 satisfies f_1. More formally:

$$f_1 \leq f_2 \Leftrightarrow \forall s \in D_H(V).f_2(s) \subseteq f_1(s)$$

If all the constant symbols, function symbols and relation symbols in the statement are interpreted as elements of H, functions on H and relations on H, then formulae can be interpreted as state predicates and statements as state transformations.

Theorem 2-3 ***Equivalent***: Two statements f_1 and f_2 are equivalent if their interpretations are identical.

There are many ways of interpreting the constant symbols, relation symbols and function symbols appearing in a WSL program. Rather than fixing on a particular interpretation, the transformation rules are proved in the context of a set Δ of assumptions. Here, Δ is a finite or countable infinite set of sentences (formulae with no free variables). In any interpretation, a sentence must either be universally true or universally false. An interpretation within which all the sentences of Δ are true is called a *model* for Δ. If S is a statement and V and W are state transformation defined by applying M to S on V and W is denoted $\text{int}_M(S,V,W)$.

Theorem 2-4 ***Semantic Refinement***: Let S_1 and S_2 be statements and V and W be state spaces such that $S_1 : V \to W$ and $S_2 : V \to W$. Let Δ be a set of sentences. If for every model M of Δ, if $\text{int}_M(S_1,V,W) \leq \text{int}_M(S_2,V,W)$ then S_2 is a refinement of S_1 under Δ and is written as:

$$\Delta \models S_1 \leq S_2$$

If $\Delta \models S_1 \leq S_2$ and $\Delta \models S_2 \leq S_1$ then the semantic functions are identical under every model, so S_1 and S_2 are semantically equivalent and is written as

$$\Delta \models S_1 \approx S_2$$

2.7.3.2 Transformation Definition

The mathematical model of WSL defines the semantics of a program as a function from states to sets of states. For each initial state *s*, the function *f* returns the set of states *f*(*s*)

which contains all the possible final states of the program when it is started in the state *s*. If two programs are both potentially nonterminating on a particular initial state, then they are equivalent on that state. A *transformation* is an operation which takes any program satisfying its applicability conditions and returns an equivalent program [112].

2.7.3.3 *Meta*WSL

A transformation is a function which maps a WSL program to an equivalent WSL program. WSL programs are represented as abstract syntax tree: therefore a transformation can be expressed as an operation on a syntax tree. Similarly, the applicability condition of a transformation is expressed as a function on syntax trees. By extending the WSL language to proved suitable constructs for accessing and manipulating WSL syntax trees, program transformations can be expressed in this extension of WSL, called *Meta*WSL. Since *Meta*WSL is an extension to WSL, the WSL transformation can also be applied to *Meta*WSL code (with some small modifications), in addition further *Meta*WSL specific transformations are possible. Therefore, a program transformation can be implemented as a piece of *Meta*WSL code which in turn can be source program for applying a transformation (including itself: a transformation can be applied to its own source code). The result will be different implementation of the same program transformation.

The implementation of *Meta*WSL involves a translator from *Meta*WSL to Scheme, a small Scheme runtime library (for the main abstract data types) and a WSL runtime library (for the high-level *Meta*WSL constructs such as **ifmatch**, **foreach, fill** etc).

2.7.4 Transformation Toolset on WSL

2.7.4.1 Related Transformation Systems

The first tool to be developed as a result of the research work on WSL and program transformation theory was the Maintainer's Assistant (MA) [111, 125]. The X windows based front-end (XMA) for MA that displays formatted WSL code provides the graphic user interface by which the user can select the position and select transformations to apply. MA includes a large number of transformations but is very much an academic prototype whose aim was to test the ideas rather than be a practical tool.

Since 1988, MA has evolved into an industrial-strength reengineering tool, FermaT [115, 118, 125], which allows transformations and code simplification to be carried out automatically. The FermaT tool was also designed to use WSL and has applications in the following areas:

- Improving the maintainability of existing mission-critical software.

- Translating programs into modern programming languages. FermaT often translates program written in obsolete assembler language to more modern languages such as C.

- Improving the quality of code by transformations.

- Extracting reusable components from the current system, deriving their specifications and storing the specification, implementation and development strategy.

- Reverse engineering existing systems to high-level specifications, followed by subsequent reengineering and evolutionary development.

FermaT provides a suitable platform and experimental environment for the program transformation step prediction approach. The proposed work is an extension of the research based on FermaT.

2.7.4.2 Implementation of Program Transformation

A transformation is function which maps a WSL program to an equivalent WSL program. WSL programs are represented as abstract syntax trees (AST); therefore, a transformation can be expressed as an operation on a syntax tree. Similarly, the applicability condition of a transformation can be expressed as a function on syntax trees. By extending the WSL language to provide suitable constructs for accessing and manipulating WSL syntax trees, the transformations are able to be expressed in this extension to WSL, called *Meta*WSL [118].

WSL syntax trees are manipulated via an abstract data type which stores the tree internally and records the 'current position' in the syntax tree. A 'position' in the tree is represented as a list of integers $< p_1, p_2, \cdots, p_n >$ where p_1 th is taken as the root, p_2 th is taken as the node at the p_2 th relatively to the p_1 th node and so on. In *Meta*WSL, a number of procedures are developed for manipulating WSL program ASTs for the implementation of transformations. For example, *Meta*WSL procedures @Up, @Down, @Left, @Right, @Goto(position), @To_Last, @To and @Down_To are used to move around the tree. The @Program function returns the whole tree, while @I returns the current item and @Posn returns the current position. The functions @GT(I), @ST(I), @V(I) and @Cs(I) return the generic type, specific type, value and list of components for the node I. Moreover, the editing procedures used to edit the current position in a tree are @Delete, @Clever_Delete, @Cut, @Paste_Over, @Paste_Before, @Paste_After, @Splice_Over, @Splice_Before, @Splice_After.

In addition to the basic procedures, *Meta*WSL also provides some basic constructs, such as **foreach**, **ateach**, **ifmatch** and **fill** constructs. With those constructs, meta-transformation can be developed and easily extended in FermaT.

2.7.4.3 Transformation Bank

Over many years of the transformation theory development, the various versions of FermaT and case studies with many different systems are developed with a large number of transformations which are known to always 'improve' the program whenever they are applied [117].

The program transformations are stored in FermaT as source files in a folder. There are two types of transformations, i.e., the 'atomic transformations' and the meta transformations'. The 'meta transformations' operate primarily by invoking other transformations. In the current transformation bank, there are nearly 100 transformations including the atomic ones and the meta ones stored as source file format.

Each transformation is written to operate on the current item @I. Hence @Trans(*trans*,data) applies transformation *trans* on @I with the given data. Most transformations do not require any data as parameters. Each transformation consists of a program called trans_name.wsl (where Trans_Name is the name of the transformation) containing [116]:

(1) An MW_PROC without parameters called @Trans_Name_Test which raises errors if the item is not suitable for the transformation;

(2) An MW_PROC called @Trans_Name_Code and taking as its parameter the data to be passed to the transformation (if any);

(3) Any auxiliary functions or procedures useful to Trans_Name_Code or Trans_Name_Test;

(4) A final SKIP which constitutes the main body of the program transformation_name.wsl, thus ensuring that technically it is indeed a well formed WSL program.

The metadata of program transformations is used to facilitate the understanding, use and management of data. For each transformation, there is an auxiliary file transformation_name_d.wsl which registers the transformation with the system and having the following form:

```
TR_Trans_Name := @New_TR_Number;
TRs_Name[TR_Trans_Name] := 'Trans Name';
TRs_Proc_Name[TR_Trans_Name] := 'Trans_Name';
TRs_Test[TR_Trans_Name]:=!XF funct(@Trans_Name_Test);
TRs_Code[TR_Trans_Name]:=!XF funct(@Trans_Name_Code);
TRs_Keywords[TR_Trans_Name] := < 'key' , 'words' >;
TRs_Help[TR_Trans_Name] := "This transformation does the following...";
TRs_Prompt[TR_Trans_Name] := " ";
TRs_Data_Gen_Type[TR_Trans_Name] := " ".
```

In addition, each transformation has a 'test file' to test its validation as well as give examples how to use it. In a 'test folder', each test file has the name trans_name_TEST.wsl where as before Trans_Name is the name of the transformation. A test file consists of a set of @Test_Trans commands. The arguments of Test_Trans consist of:

- A string identifying the test: '*n*th test of Trans_Name';

- The item to be tested;

- The position in this item at which the transformation should be applied (given as a list — see Section 2.7.4.2);

- The data for the transformation: if there is none, the data will be an empty set;

- Either the code which should result from the transformation (if the item to be transformed ought to pass Trans_Name_Test in trans_name.wsl) OR the string 'Fail' if it should be failed by Trans_Name_Test.

2.8 Summary

The chapter introduced the techniques and existing research which are relevant to the proposed approach.

- The goal of software reengineering is to understand the existing software and then to re-implement it to improve the system's functionality, performance or implementation.

- The process of reengineering computing system involves three main steps, i.e., reverse engineering, restructuring and forward engineering.

- Program transformation is the act of changing the syntax of a program but preserving its semantics.

- Program transformation has applications in many areas of software engineering. It is one of the important techniques for software reengineering due to its features.

- A number of transformation systems are reviewed with regard to their platforms, processing languages and capability. Compared to the other systems, FermaT platform is an industry intensive toolset for software migration and software reengineering. Equipped with formal techniques and the rich transformation bank, FermaT has a good extendibility and it is ideal to experiment the proposed transformation prediction approach.

- Software goal-driven techniques and software metrics reviewed in this chapter are appropriate for the representation of target driven by a reengineering activity.

- A variety of the related work on program transformation automation and using heuristics based search techniques to get optimal solutions to achieve the automation are reviewed and analysed. The defects of these approaches are discussed and accordingly a few points should be addressed and paid attention to in the proposed approach.

- The background and concepts of WSL is introduced as the groundwork on which the language and transformation extension for further applications is flexible due to its language levels.

- The current transformation bank is reviewed on the aspects about its capacity, management and development. It provides a good platform to explore the transformation prediction approach.

Chapter 3
Target Driven Transformation Step Prediction Framework

Objectives

- To outline the motivation of the Target Driven Program Transformation Step Prediction (TDPTSP)
- To explore the relationship between targets, metrics and transformations
- To overview the framework of the proposed research
- To illustrate the technical steps used in the proposed work

3.1 Introduction

As a combination of reverse engineering and forward engineering, software reengineering is the set of activities for the problem of evolving existing computing systems to ensure that software continues to meet organisational and business targets in a cost effective way. Each activity is triggered by specific requirements or targets. To achieve the targets proposed for the software reengineering, program transformation, also called *transformation* for short, can be used for the specified purposes. The

maturity of Wide Spectrum Language (WSL) and its transformations has reached the point which has the large scale of industry applications [111, 118, 125]. A large number of transformations stored in the transformation bank can serve the reengineering tasks and result in the desired software which fulfils the specific reengineering targets. The efficiency of finding the proper transformations and implementing them for the specified targets has become a very big concern of the maintainers. The thesis proposes a method to predict the transformation steps and aid transformation process to approach the reengineering targets. This chapter aims to overview the framework of the proposed approach and introduces its technical steps.

3.2 Problem Definition

3.2.1 Program Transformation for Software Reengineering

Software reengineering is important for recovering and reusing existing software assets, putting high software maintenance costs under control and establishing a bass for future software evolution. As defined by Tilley [105], reengineering is the systematic transformation of an existing system into a new form to release quality improvements in operation, system capability, functionality, performance or evolvability at a lower cost, schedule or risk to the customer. Reengineering is an umbrella term which covers many forms of system improvement, many of which are tool supported. Starting from existing source code, software reengineering is combined of reverse engineering, restructure and forward engineering. The process of reengineering aims to lead software evolving [47] successfully through a set of 'Re'-activities, listed in Chapter 2.

A program transformation refers any operation on a program which generates a semantically equivalent program. Program transformation is used in many areas of software engineering, including compiler construction, software reverse engineering.

software vaiualisation, documentation generation and automatic software renovation [93]. With its capability in those areas, it has been one of the most important and functional techniques for software reengineering. The program transformation has been proved to be a powerful technique for deriving programs from specifications, verifying program properties, improving source code quality, comprehending source code, specialising program with regard to their context of use, deriving more efficient program versions from less efficient ones and so forth [91]. The research and development of program transformation in both academic and industrial areas has been booming since it was proposed [20].

3.2.2 Program Transformation with WSL

As an important member of the program transformation family, the WSL and the transformation theories based on the language have been developed and used in the both areas. It was developed as an academic prototype when it was 'born' in 1988 [111]. Since then the great efforts are devoted into its development and research. It has become more and more mature and been used in large scale industrial projects [115, 117, 118]. The transformations based on WSL are implemented in a tool named *transformation engine*. The latest version of the transformation engine is called FermaT [114, 118] and forms a central component of FermaT Workbench [115]. The transformation technology of FermaT has reached such a level of maturity that behaves on the following perspectives [125]. The tool can cope with the usual programming constructs and their uses.

In the library of the transformation engine, there have been nearly a hundred transformations developed. The transformations in the library were proven correct before the tool was built. They allow a construct in WSL to be recast into another WSL

construct while ensuring that the semantics are preserved.

The software engineer does not have to do the proof and only has to select a transformation and apply it by using the tool. The transformation engine checks that if the transformation is applicable. The whole transformation process, from the raw WSL generated directly from the parsed source language to high-level WSL suitable for the translation to the target source code, could be carried out automatically with no human intervention.

Since transformations are implemented in *Meta*WSL, it is possible for users to develop their own transformations (as combinations of existing transformations) and add them to the system. Provided the new transformations are limited to invoking existing transformations and are prevented from carrying out unrestricted editing operations, the new transformations are guaranteed to be correct.

3.2.3 Traditional Semi-Automatic Program Transformation

Before the motivation of the proposed approach is introduced, the traditional execution of program transformations is reviewed to show the routine steps included in the normal program transformation process. Traditionally, the execution steps of transformations are predefined by the software engineer according to his/her expertise and the usage of the selected transformations. After selecting the transformations and deciding their sequence, the reengineering task can be implemented by following a batch file which contains the determined information. For instance, the following example shows how to migrate Assembler code to C code via the transformation on WSL.

```
1   a2w "FMT001A1.lst" "FMT001A1.ws1"
2   metrics "FMT001A1.ws1" "FMT001A1.me1"
```

```
3   datprune "FMT001A1.ws1" "FMT001A1.dat" "FMT001A1.da1"
4   datreorg "FMT001A1.da1" "FMT001A1.da2"
5   dat2ll "FMT001A1.ws1" "FMT001A1.da2" "FMT001A1.ll"
6   dat2c "FMT001A1.da2" "FMT001A1.h"
7   dotrans "FMT001A1.ws1" "FMT001A1.ws2" Find_Dead_Code
8   dotrans "FMT001A1.ws2" "FMT001A1.ws3" Data_Translation_A data="FMT001A1.ll"
9   dotrans "FMT001A1.ws3" "FMT001A1.ws4" Fix_Decimal data="FMT001A1.ll"
10  dotrans "FMT001A1.ws4" "FMT001A1.ws5" Fix_Assembler data=25600
11  dotrans "FMT001A1.ws5" "FMT001A1.ws6" Data_Translation_A data="FMT001A1.ll"
12  metrics "FMT001A1.ws6" "FMT001A1.me6"
13  wsl2c "FMT001A1.ws6" "FMT001A1.raw"
14  tidy_c "FMT001A1.raw" "FMT001A1.c"
15  gcc -i . -i /home/martin/fermat2/config -c "FMT001A1.c"
```

Steps 1-6 are the processes before the code improving transformations. The preprocess aims to translate the Assembler program to WSL, measure the raw WSL code and restructure the data file. Steps 12-15 are the processes after the transformations. These steps are used to measure and translate the processed WSL code to the target source code.

Steps 7-11 are the actual processes to apply the transformations. The first step determined by the maintainer is to find the dead code, i.e. unreachable code in the program. The next transformation Data_Translation_A is to load the data file. The Fix-Decimal transformation is specially to deal with the assembler system's feature. The transformation Fix-Assembler is used to transform the action system to the procedural structure with IF/While structure, then reload the data file and generate the final transformation result.

The transformations are executed one by one while their sequence is predefined. It is a semi-automatic process where the execution is decided by software engineer. To find the proper transformations and determine their steps greatly depends on the software engineer's experience, understanding of the existing transformations and the source

code features.

However, it is quite often that a recruit could not define the sequence due to the lack of the expertise. On the other hand, to understand the features of the program is a time-consuming mission. Furthermore, without the knowledge of the existing transformations also can cause the failure of the transformation. These problems drive the needs of the proposed research.

3.2.4 Need to Predict Program Transformation

It has been learned that experience often is employed as a pattern of practice. In practical work, the choice and implementation of transformation sequence mainly depends on the experience of software engineer. It is required that the engineer has to read the WSL program and understand the source code features first. The success of the process heavily relies on the expertise and domain knowledge of the software engineer. Such dependence makes the transformation process short of efficiency and correctness if the transformation is used inappropriately. Without the experience, the transformation selection could not be performed. This is a common problem for a recruit to use the tool. In addition, there is another crucial factor for the selection of the transformations, i.e., the purpose of transformation. As discussed above, the transformations are applied for specific purposes, called targets during software reengineering. The satisficed degree of the targets needs to be measured properly. Normally, this measurement is made by the engineer's estimation which could lack of accuracy and efficiency.

Therefore, an intelligent feature of the transformation engine, which can guide the transformation process smartly, is required. How to develop techniques to improve the automation and efficiency of program transformations is a concern in both research and

industry area [3, 4, 41, 104, 130, 134]. The proposed research aims to achieve the goal by providing an appropriate mechanism to predict the suitable transformations then consequently the efficiency and correctness for the target achievement can be enhanced. A contribution of the thesis is to research and develop a prediction mechanism. The transformation prediction can become a feature of the current transformation engine to assist and present clues to the users and accordingly to improve the transformation implementation's efficiency. To provide useful information guiding the transformation process, the transformation engine is supposed to determine the suitable candidates and predict the sequence of the transformations.

To develop such an intelligent predictor for transformation, there are three essential factors, reengineering target, quantitative measurements of software and existing transformations. In this thesis, the formal representation of reengineering targets and metrics are correlated into a model by referencing existing techniques of requirement representation [83] and measurement [89]. With the target based quantitative information related to the program features, it is possible that the transformation engine can smartly predict the candidates and their execution solution within the transformation bank and perform the program transformation in an efficient way.

To summarise the need to predict the program transformation, the following points are listed.

- To regard the experience of implementing transformations as heuristic knowledge for assisting the transformation activities
- To facilitate transformation process with domain knowledge
- To promote the automation of the transformation

- To predict the transformations for specific targets

- To utilise target and its measurement as a knowledge to assess the impact of the transformations

- To improve the efficiency of transformation process to approach the specified targets

3.2.5 Need to Extend WSL

Software engineers can take advantage of WSL as an intermediate language to achieve various reengineering purposes. The current transformation applications are mainly related to unstructured code, such as Assembler, or procedural High Level Language (HLL), such as COBOL or C. The existing layered structure of WSL determines that WSL can only be utilised for such kinds of applications. In most practical cases, there is also high demanding for analysing different types of programs, such as object-oriented programs.

In the related previous work [72, 78] , the extension of WSL with object orientation was performed partially. However, their work only provides some new syntax constructs, such as to declare a class only, which are not proposed based on the WSL levels including both of the semantics and the syntax aspects. Besides, they did not define the precise semantics for those new constructs so that the semantics-preserving transformations based on those new constructs cannot be proved and performed either.

In addition, the domain features are not taken into account when program transformations based on WSL are performed. In practice, the domain features are important knowledge for software reengineering. An aim of the proposed work is to

utilise the domain features in the transformation process. For this purpose, the multimedia domain is studied as an experiment. The study needs the extension of WSL itself with the domain features. Correspondingly, the transformations based on the language extension are extended.

To summary the need to extend WSL and the transformation bank, the following points are listed.

- To extend WSL with more language features, such as object oriented features.
- To extend WSL with domain features.
- To extend WSL on both of the semantics and the syntax aspects.
- To give the precise semantic definition of the extended language.
- To extend the transformations based on the extended language.

3.3 Reengineering Targets, Metrics and Transformations

3.3.1 Definition of Reengineering Target

The targets of software reengineering through program transformation normally are straightforward but abstraction. They can be described as quality improvement requirements, such as 'having high performance' and 'improving the understandability'. These non-functional targets are called soft-goal in [130, 134]. The other kinds of targets which may involve in the software reengineering are also taken into account. For example, abstraction is an important aspect for program comprehension. The target to improving the abstraction of the source code could be

described by a variety of criteria.

Definition 3-1 (Target) a target τ is an objective that the reengineering action under consideration should achieve. Targets are optative statements as opposed to indicative ones and bounded by the subject matter.

Definition 3-2 (Target Factor) A target factor t refers to an intended property to be ensured. It is bounded and associated with the other target factors altogether to support satisficing the target τ. Each factor can be measured by one or more metrics. Alternatively, a target factor is a sub-target of the target τ.

According to the definitions given above, a target might be affected by a set of factors. For example, the target to decrease the complexity of the software could include a few factors, such as component size, program complexity, structureness and component nesting level. The definition of 'target factor' is more specific than the one of 'target' so that a target can be shifted from an abstraction level down to a specific level. The unit of a target can be normalised to a united range by utilising the target factors.

3.3.2 Relations between Targets, Metrics and Transformations

The target driven approach proposed in the thesis implies natural relations between transformations, targets and metrics. Figure 3-1 illustrates these relations.

The diagram consists of two parts, i.e. the *Process* part and the *Targets* part. In the *Process* part, every transformation can affect the code after it is implemented. The impact of a transformation can be measured by one or multiple metrics which are related to the specific targets. The *Targets* part describes the motivation of the *Process*. In the beginning, the motivation is to pursue the specific targets and at the end, the

result should be to approach the target as much as possible.

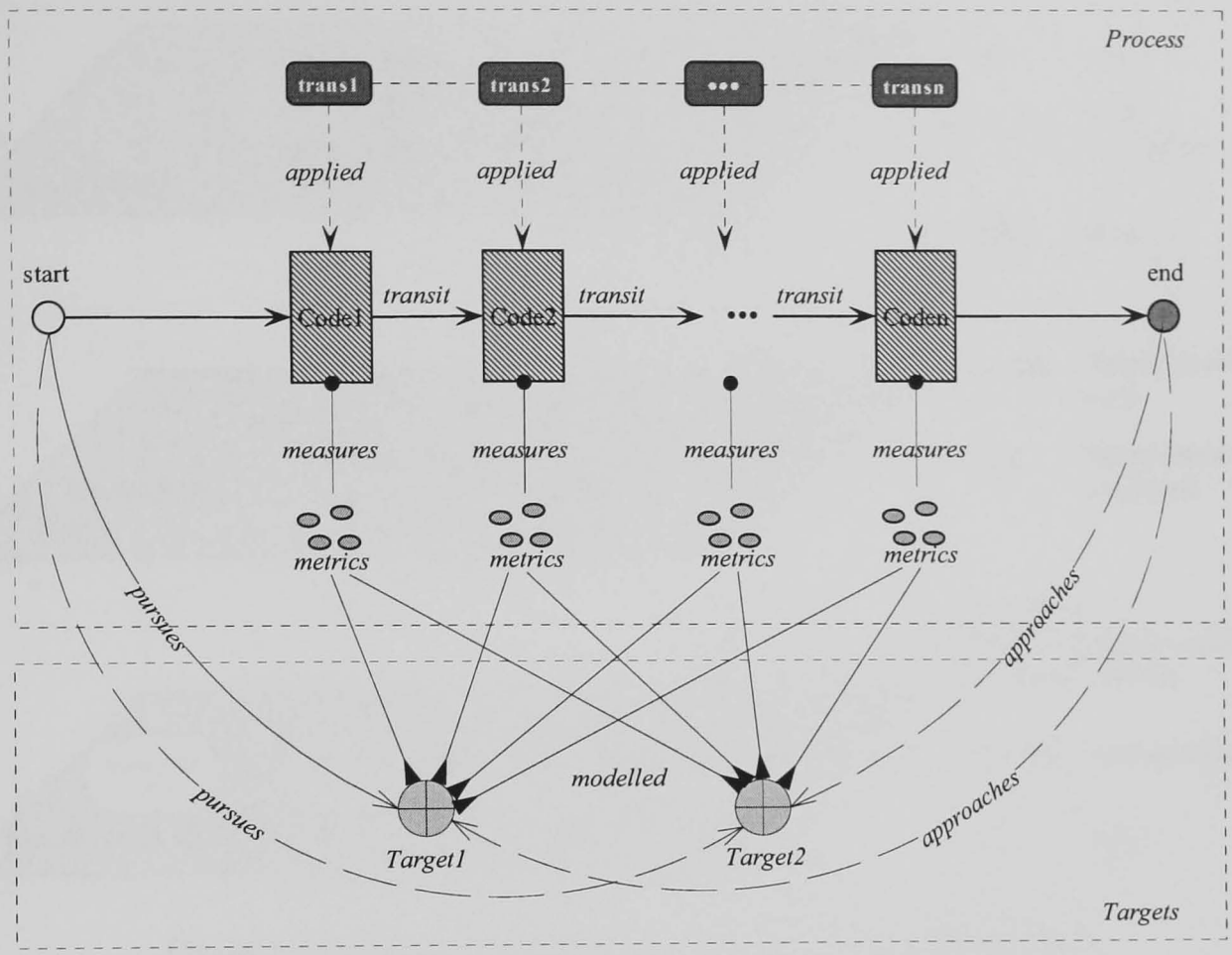

Figure 3-1 The Relations between Transformations, Targets and Metrics

To present the relations more clearly, Figure 3-2 gives a combined Model for Target-Metric-Transformation (MOTMET), which displays the different models as three layouts according to the entities in the model.

The top level is the most abstraction level which modelles a reengineering target in a Target Model (TM). In Chapter 4, the method of the model construction and the computation related to target will be presented. The calculation based on the target computation will be used to evaluate the impact of the transformations.

The middle level is the implementation level containing the transformation process model, which results in the change of source code at the bottom level where the changes over versions of source code are raised by the transformation implementations and

measured by the metrics correlating to the target.

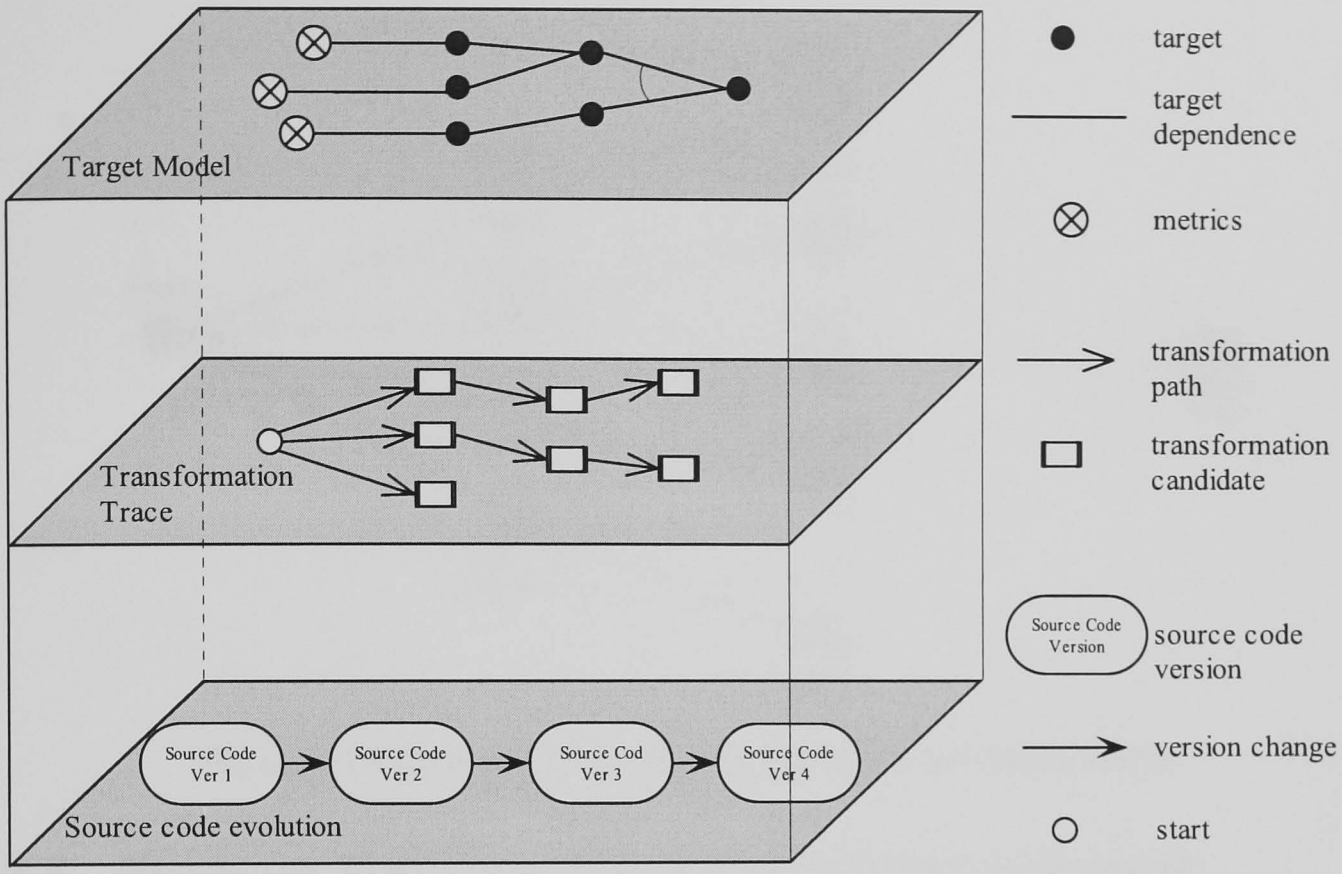

Figure 3-2 Model for Target-Metrics-Transformations (MOTMET)

The combined model gives an overview of the interrelations between the models as well as the means to predict the transformation steps for software reengineering. How to use and implement the model in the proposed approach will be elaborated in the next section.

By the analysis of the relations depicted as the above figure, it can be concluded that a suitable solution to implement transformations can facilitate the approach to targets efficiently. Therefore, to find the solution is crucial for the determined task. Figure 3-3 simulates a paradigm to find possible transformation solutions. The solution is called as Transformation Path (TP) within a Transformation Process Model (TPM). The problem to find the *TP(s)* can be modelled as a search problem which is used to construct TPM. Each node represents a state of program version. The nodes are valued by a vector

which contains the transformation steps and the quantitative measurement value. The details will be presented in Chapter 6.

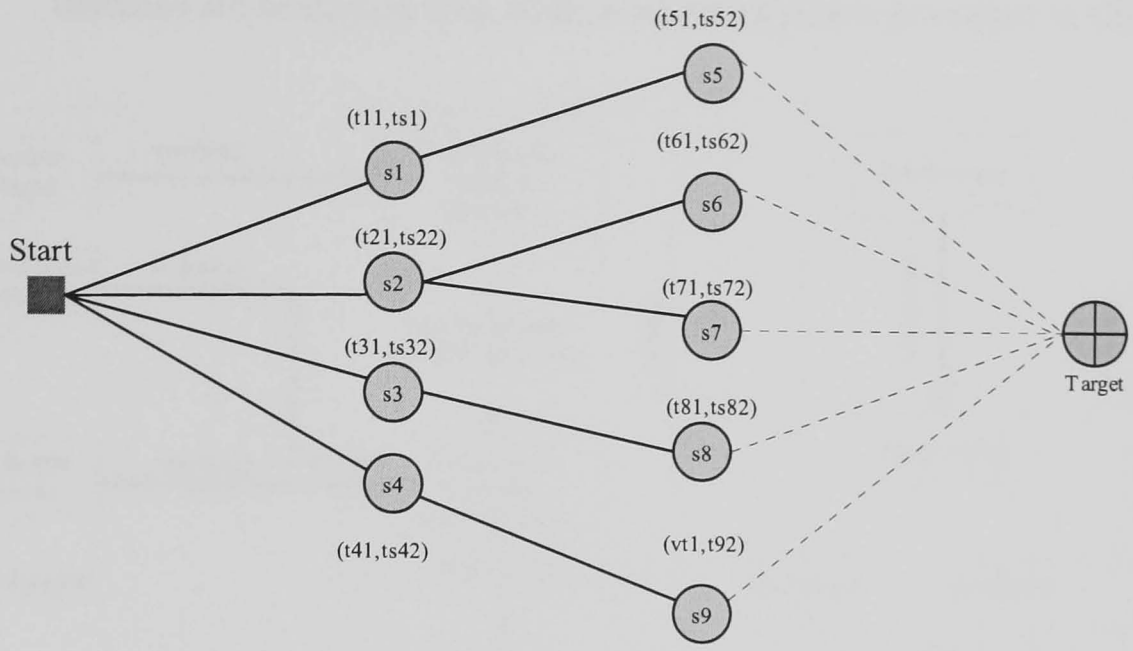

Figure 3-3 A Paradigm of the Transformation Process Model (TPM)

3.4 Program Transformation Prediction Framework

3.4.1 Target Driven Transformation Prediction Framework

To describe the proposed approach further, the technical steps in the approach are presented in a framework called Target Driven Transformation Prediction Framework (TDTPF) is proposed as the mechanism and shown in Figure 3-4. In the framework, the extension of WSL and the process of the transformation predication are described as follows.

(1) Based upon the basic levels of WSL, the secondary level of WSL are added specific features, such as object-oriented feature or domain features. The extension is performed on both semantics and syntax and developed in the transformation engine. In addition to the extension with object-oriented features, the specific domains, such as multimedia domain is studied as an

instance. Along with the extension of the WSL language, the definition of semantics and the behaviour-preserving transformations for corresponding domains are presented. The work of the extension is described in Chapter 5.

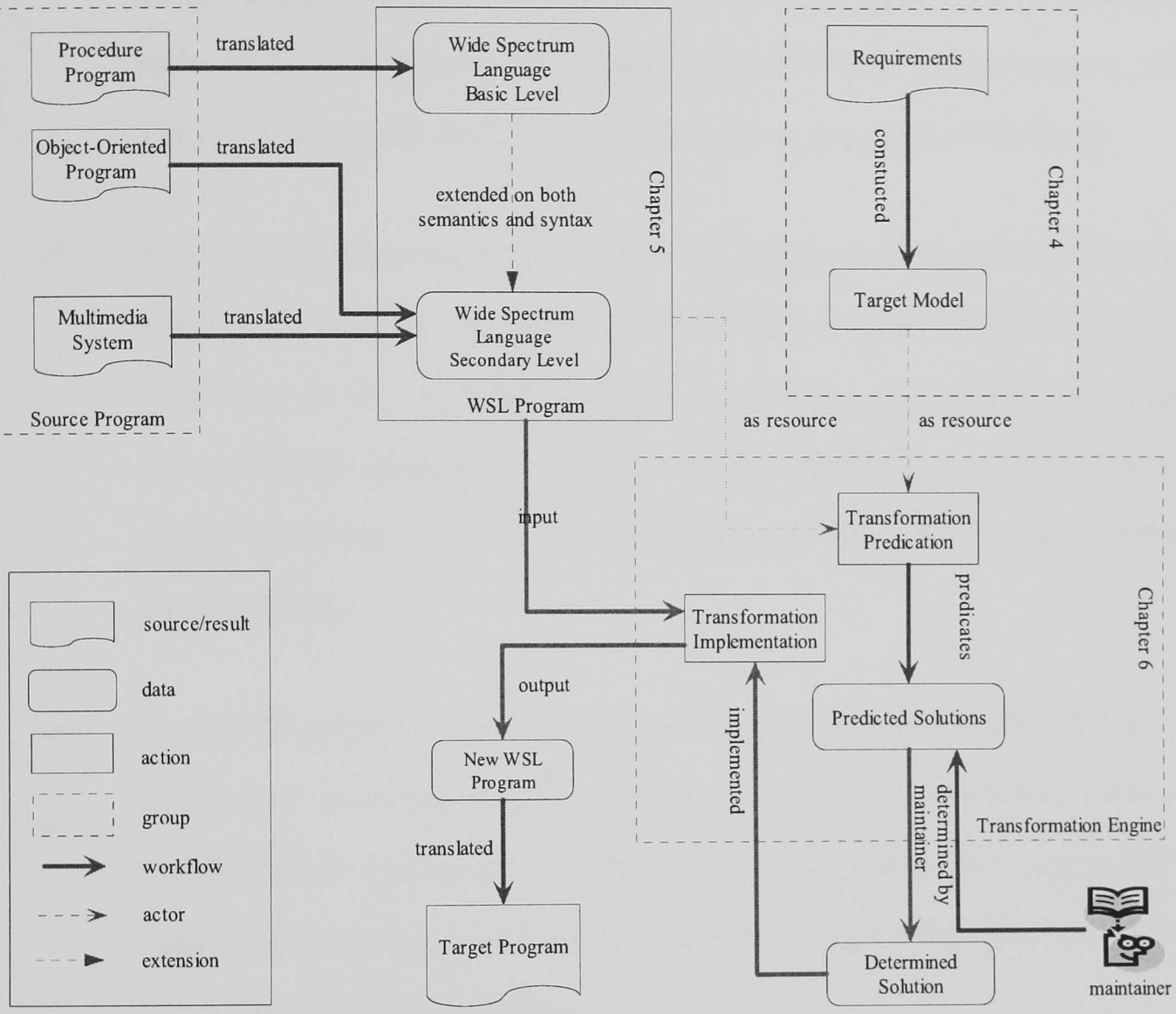

Figure 3-4 Target Driven Transformation Prediction Framework

(2) Before having transformations applied, the source program is translated to a WSL program according to the corresponding syntax of WSL. This process is performed by a translator between source program and WSL which is used as an intermediate language.

(3) By getting the translated WSL program, the transformation engine reads a WSL program and the desired targets as the resources. The targets are

modelled as a Target Model (TM) which contains the detailed target factors affecting the targets. The TM is constructed formally by using the existing requirement modelling approach. The leaf nodes of the model are measured by the selected metrics. The metrics are used as the quantitative means to scale the satisficed degree of the desired targets. In the implementation, the TM is represented in XML format. The work is described in Chapter 4.

(4) The transformation engine pre-checks the source code and determines the source code features. Pre-check is a necessary step for the transformation prediction. At the pre-check step, the target model and the WSL program are used as the resources. By referencing the resources, the transformation engine constructs a model for predicting the solutions for the desired transformations.

(5) During the prediction process, a Transformation Process Model (TPM) is constructed according to the program. The heuristics obtained from a metrics based algorithm, or pattern based algorithm that incorporates expertise and domain features are essential to construct the TPM. The work is described in detail in Chapter 6.

(6) The heuristic knowledge comes from common knowledge for applying program transformation or expertise from the practical work. According to the knowledge, the heuristic function is constructed and calculated by given formulae or rules. This work is illustrated in Chapter 6.

(7) The TPM is a tree-structured diagram and composed of Transformation Paths (TP). Each TP is expanded by three factors, i.e., the applicability of a transformation, heuristics knowledge, which implies the relation between a

transformation and the target, and measurement of the transformation impact. When a node is added into the path, it will be valued by a formula to evaluate the transformation's impact. At this step, the heuristics are used to search and determine the transformations to be selected. This work is presented in Chapter 6.

(8) As a TPM is constructed, the ranked transformation paths in the model are generated and listed for software engineer as solutions. This work is presented in Chapter 6.

(9) After obtaining the TPs, software engineer will be the one to determine the final solution based on the predicted information. By comparing the impact of the transformations, the software engineer decides to choose the solution and sends to transformation engine to perform the solution. This work is presented in Chapter 6.

(10) The transformation engine executes the selected determined solution including the preferred program transformation steps and output the generated new WSL program. The process to predicting and executing transformations is repeated until the result is validated and satisfied by the software engineer.

(11) The transformed and validated result is translated to the original language or another targeted language according to the needs of the scenario. The final translated result will have the desired properties which fulfil the targets.

3.4.2 Dedicated Metrics

Software metrics are used to measure the software status and play the role as coordinators between targets and transformations. In the thesis, a set of metrics are selected and classified as six groups according to their characteristics. The six groups of the metrics can be chosen and composed to serve desired target(s). The selection and composition is performed based on the target model which will be exploited in the later chapter. The dedicated metrics are reengineering-intensive ones and involved in complexity metrics, abstractness metrics, object-oriented metrics, reusability metrics, domain specific metrics and feature oriented metrics.

3.4.3 Correlation of Metrics to Reengineering Targets

According to the definition of the target and target factor, the satisficed degree of a target can be measured by the correlation of metrics. The metrics are chosen and correlated together from the developed metrics for the specified target. The target formula describes how to correlate the metrics. The target score of a target is calculated by the correlated metrics and the defined relations. The correlation formula is given according to the target model which is defined by the goal driven techniques. The correlations can be depicted as AND/OR relations.

3.4.4 Automating Transformation Steps

A key point of the transformation prediction is to automate the transformation process towards the desired target. The automating process can be driven by the following elements:

- The targets

- The dedicated metrics
- The transformations
- The state of the program

The automation of the process behaves in the following aspects:

- Transformation determination
- Transformation steps

3.4.5 Incorporating Expertise

Only by the assistance of the metrics correlated in a target model, the predicted result could be so objective that too many candidates could be captured. Expertise is a crucial factor in software analysis, especially in software reengineering, in that it provides an efficient means to understand and give the hints to analyse software. Once incorporated expertise, the prediction mechanism can be more appropriate due to the importance of the expertise in practice. The transformation steps generated can be determined by the software engineer who has the expertise for program transformation and reengineering. Therefore, the expertise can be regarded as a kind of heuristic knowledge used in the prediction process.

3.5 Summary

In the chapter, the motivations of the proposed approach are discussed and the technical steps of the approach are illustrated. To recap, the purpose of the research is to develop an appropriate mechanism to guide program transformation process towards reengineering target with the assistance of quantitative means and heuristic knowledge. As an overview of the proposed work, the chapter addressed the following points technically.

- Compare to the traditional program transformation process which is a semi-automatic procedure characterised by the predefined transformation sequence, the proposed research goal is to improve the automation and efficiency of using transformations.

- The purpose of a reengineering activity through program transformation is called target which is modelled with quantitative measurements, such as software metrics.

- There are tight relations between targets, metrics and transformations. The work is performed based on these relations to achieve the research goal.

- The proposed approach is called TDPTSP, taking the modelled target and WSL source code as input and giving the predicted transformation steps as output.

- The proposed approach is explored in specific domains; therefore WSL will be extended to adapt these cases.

- For the target driven program transformation process, the transformation bank needs to be extended as well.

The following chapters will exploit the details of the framework introduced in this chapter.

Chapter 4
Using Software Metrics to Describe Reengineering Targets

Objectives

- To propose and justify six categories of software metrics
- To address the relationship between metrics and reengineering targets
- To define and formally model reengineering targets
- To propose the formula for measuring a specific target

4.1 Introduction

Software metrics can be used throughout the software life cycle to assist in cost estimation, quality control, productivity assessment and project control and can be used to help assess the quality of technical work products and to assist in tactical decision making as project proceeds [125]. Reengineering includes reverse engineering and forward engineering in order to achieve different objectives, such as improving the quality of software or changing the code to a higher abstraction level. These objectives, which are called reengineering targets in the thesis, can be measured and evaluated by

given metrics as the normal engineering disciplines. This chapter aims to propose and justify the reengineering-intensive metrics, investigate the usage of metrics to measure the reengineering targets and associate the metrics to the targets by a given model.

4.2 Software Metrics for Reengineering

Software metrics are used to quantify particular characteristics of software systems. By combining the use of software metrics, the reengineering process is guided towards the identified targets in the specific stages. Software metrics for forward engineering have been developed maturely while for reverse engineering are a much neglected area. Concrete measures for reverse engineering can be developed hierarchically and revised from the measures which are for forward engineering. Numbers of selected measures for forward engineering can be adapted for reverse engineering. Then new software measures for reverse engineering can be developed, based on existing measures for forward engineering or from scratch [125]. Zhou et al. [132] proposed five categories of measures for reverse engineering. The five categories can be referred and adapted in the proposed approach for measuring the reengineering targets.

4.3 Six Categories of Software Measures

In the thesis, six categories of software measures are selected. These measures are classified according to their characteristics and used to guide the reengineering through program transformation process. The choice the six categories of software measures relies on the following reasons.

- The six categories of software metrics are selected to measure the essential aspects of reengineering.

- Improving the internal structure of a program is a mission of program transformation. The internal attributes of program contribute a crucial role to most quality requirements. In both reverse engineering and forward reengineering, complexity measure plays a crucial role for most of quality measurement. Hence, complexity metrics are selected for non-functional reengineering target measurement. As for object-oriented program, the object-oriented metrics are necessary for measuring its internal structure.

- As a vital part of reengineering, reverse engineering emphasises raising program representation from the low level of abstraction to the high level. Without tackling abstractions properly, any design or specification recovery method cannot succeed. Therefore, the metrics to measure abstractness are included in the consideration of metric selection.

- Reusability can be regarded as a kind of external property and is an important motivation of reengineering. By using the reusability metrics, it is possible to measure the likelihood of utilising the existing software and pursue the further reengineering.

- The application of program transformation on domain specific applications, such as multimedia is an extension to the traditional usage of reengineering through transformations. To measure this process, the metrics to measure the status of a multimedia application are taken into account.

- Feature oriented reengineering has been emerging as a proficient means for software comprehension and software alteration. As an advanced metric category, feature oriented metrics are used to facilitate the program transformation process driven by the feature oriented targets.

4.3.1 Complexity Metrics

In most cases, the targets pursued in software reengineering have to conform to hard and soft quality constraints (or non-functional requirements). These desired qualities (or, more precisely, desired deltas on these qualities) play a fundamental role in defining the reengineering process and the tools that support it.

Quality requirements of a system are attributes and characteristics of the system. Several approaches such as IEEE Standard [55], ISO 9126 [57] and ISO 9421 [56] etc. dealing specifically with quality requirements have emerged. International Organisation for Standardisation (ISO) introduced taxonomies of quality attributes [57] which divides quality into six characteristics: functionality, reliability, usability, efficiency, maintainability and portability.

The above models classified the quality requirements at a high level and presented factors or criteria for each quality. The internal attributes of program, such as modularity and complexity, contribute a crucial role to most quality requirements. On the other hand, a strongpoint of program transformation is the possibility of improving the internal structure of a program. Therefore, the internal properties of program, i.e. source code quality, are paid more attention. Complexity is one of the most pertinent characteristics of computer software. In forward engineering, complexity measures are mainly used to indicate the quality of software. In a reverse engineering process, people mainly want to understand an existing program through reverse engineering from the original program to less complex specifications, because the less complex a program is, the easier it is for people to understand it. Low complexity is an important factor that results in the high values for qualities, such as reusability and maintainability. The complexity metrics chosen for the proposed research is shown in Table 4-1.

Metrics	Definition
NCNB	The non-comment non-blank number of statements in the program [125]
McCabe	The number of linearly independent circuits in a program flow-graph. This measurement is calculated as number of predicates plus one [74].
WOC	The sum of the weights of every construct in the program. The construct is defined subjectively according to experience gained by engineers and managers as show in Table 4-1 (a) [125].
NON	The number of nodes in the abstract syntax tree [125].
CFDF	The number of edges in the control flowgraph (CF) plus the number of times that variables are used [125]
RNC	The number of instances of recursion and nesting in the program [125]

Table 4-1 Selected Complexity Metrics

Construct	Weight	Construct	Weight	Construct	Weight	Construct	Weight
+	1	=	0	>=	0	IF	4
-	2	<>	0	Min	1	While	4
*	2	>	0	Max	1	Do	10
/	3	<	0	Div	2	D_IF	10
**	3	<=	0	Mod	2	Abort	2

Table 4-1 (a) Sample Weight Values of Constructs

- **Component size** is used to evaluate the ease of understanding of code by developers and maintainers. Size can be measured in a variety of ways. Non-comment Non-blank (NCNB) is sometimes referred to as Source Lines of Code and counts all lines that are not comments and not blanks. In this context, small source code size relates to low complexity and therefore leads to high reusability, understandability and maintainability. The metric Number of Node (NON) is also used for measuring the component size.

- **Program complexity** is fundamental to reduce overall program complexity

and enhancing quality. There are two ways to quantify method complexity: information flow and internal control structure. Information flow relates to complexity as measured by the number and types of formal parameters, as well as the number of method invocations. The more control and data flows a method has, the harder it is to be modified and consequently the harder it is to be reused. Similarly, the internal control structure of a component relates to the complexity of the control flow graph and it is measured by the McCabe complexity [43]. The fan in – fan out complexity which maintains a count of the number of data flows from a component plus the number of global data structure that the program updated.

- **Structureness** is the sum of the weights of every construct in the program. The construct is defined subjectively according to experience gained by engineers and managers.

- **Control-Flow and Data-Flow complexity (CFDF)** is the number of edges in the control flowgraph (CF) plus the number of times that variables are used (defined and referenced) (DF).

- **Recursion and Nesting Complexity (RNC)** is the number of instances of recursion and nesting in the program.

4.3.2 Abstractness Metrics

Both abstraction and refinement are used to enhance analysis and optimise program. A refinement is an operation which modifies a program to make its behaviour more defined and/or more deterministic. The opposite of refinement, abstraction, is a process of generalisation, removing restrictions, eliminating detail, removing inessential

information (such as the algorithmic details). Without tackling abstractions properly, any design or specification recovery method cannot succeed. In a broad sense, abstraction corresponds to weakening in semantics and this weakening is due to the following [72]: (1) Inessential design/implementation details are omitted; (2) Non-determinism is increased; (3) 'How to do' is substituted by 'What to do'.

Within WSL supported environment, such as FermaT [118], refinement and abstraction are performed as transformations for both forward and reverse engineering.

Metrics	Definition
ABST-LOC	The quotient of Lines of Code (LOC) over the number of nodes (NON) in the abstract syntax tree.
ABST-CFDF	$ABST-CFDF=\frac{1}{n_1+n_2}$, where n_1 is the number of times variables are referenced in procedures and function; n_2 is the number of times that variables are defined.
ABST-VOC	$ABST-VOC=\frac{B_i(SC)}{B(SC)}$, the percentage of constructs at higher abstraction levels in the total constructs in the programs.

Table 4-2 Selected Abstractness Metrics [124]

Due to the importance of abstraction, abstractness target is chosen to gear a program into a proper abstraction level. To gather if the program is 'abstraction' enough to capture the right 'specification', the abstractness target can be measured by abstraction measures proposed in [124, 125, 132]. The measurement of abstractness is presented in Table 4-2.

4.3.3 Object Oriented Measures

Object orientedness is the degree to which a system or its components has a design or implementation that is expressed in terms of objects and messages via encapsulation, inheritance and polymorphism between the objects. Following the strong trend toward object-oriented technology, object orientedness measures have become an unavoidable subset of software metrics and engineers want to reengineer their huge number of conventional procedural systems into object-oriented systems.

Metrics	Definition
NMI	The number of method invocations
DIT	DIT is the length of the longest path from the class to the root in the inheritance hierarchy [23]
CBO	As for a class, it is the number of other classes to which it is coupled. It relates to the notion that two classes are coupled when methods in one class use methods or instance variables defined by another class.
WMC	Wight of Methods in Class (WMC). Consider a class C_1, with methods $M_1,\ldots, M_n$ that are defined in the class. Let $c_1,\ldots c_n$ be the complexity of the methods, then $WMC = \sum_{i=1}^{n} c_i$ if all method complexities are considered to be unity, then WMC = n, the number of methods. Here the number of methods is calculated as the summation of McCabe's cyclomatic complexity of all local methods.
NVC	NVC is the average number of public variables and private variables per class.

Table 4-3 Selected Object Oriented Metrics [125]

In reengineering, the object oriented (OO) measures give the complexity of classes and relationships between classes. OO measures can be used to measure source programs, transitional program and specifications, which can help effective and efficient reengineering of OO system. The existing OO measures for forward engineering can be adapted for reverse engineering. These metrics selected are listed below for measuring OO system in the reengineering process. In Table 4-3, the dedicated OO measures are chosen in the proposed approach.

4.3.4 Reusability Metrics

The main attributes of reusability are generality, transportability and retrievability. Understandability is also reflected by the feature of reusability. Generality measures estimate whether the system or components of the system perform a broad range of functions so that they can be used in more than one computer program or software system. Transportability measures also give the ease of translating programs. Retrievability refers to the ease of design recovery.

Metrics	Definition
WOIL	Weight Of Interfaces in relation to Lines of code: this is a measure calculated by dividing the number of lines of interface code by the total number of lines of code.
HIL	Human Interaction Level in relation to lines of code: This calculates human action level by lines of commands. $HIL = \frac{Lines\ of\ users'\ command}{Source\ lines\ of\ code}$
AMS	Average Module Size: This measure is calculated by number of statements over number of methods. $AMS = \frac{LOC}{Number\ of\ methods}$
SD	Self-Descriptiveness: This estimates the weight of on-line comments and statements with the self-descriptiveness characteristic in the program. $SD = \frac{Number\ of\ on-line'\ comments\ and\ special\ statements}{Total\ number\ of\ statements}$
ETL	Error tolerance level (ETL): This measures the weight of parts in the program that can be used to detect errors and remind errors. $ETL = \frac{Lines\ of\ error'\ indentifying\ components}{Source\ lines\ of\ code}$

Table 4-4 Selected Reusability Metrics [125]

A main stream of reengineering is to reuse the existing software assets to accommodate the requirement change. Depending on the measures of reusability, a software engineer can know what should be done after reengineering. Reusability measures are always used to measure resources and initial products in the initial stages of the reverse engineering or the final stages of forward engineering. Normally, reusability measures will not be supposed to measure the process and transitional products in reengineering. Several reusability measures [125] are listed in Table 4-4.

4.3.5 Domain Specific Metrics

Domain features are crucial knowledge when an application is reengineered. Normally, these features can behave in a number of aspects, such as program language, data, control, measurement of the application and so forth. In the proposed transformation prediction framework, the domain features are processed by four means:

- Source code in extended WSL which is augmented with the domain features
- Domain specific targets which concerns the characteristics of the domain
- Measurement of the domain specific application, i.e., software metrics
- Transformation prediction process assisted with the domain features

The reengineering targets identified in the specific domain could be different from the reengineering targets in the general domain. Due to the difference, the measures attached to the targets could be special compared to the common metrics. The section will investigate the measurement in a specific domain, using the multimedia domain as an example. In the later chapter, the further processing of domain features for transformation prediction will be elaborated.

■ **Measures in Multimedia Domain**

Multimedia languages are a new class of languages that have arisen in the past decade. The term 'multimedia' refers to "a presentation or display that involves more than one method or medium of presentation" [73]. Such media may include audio, video, still images and animations that accompany the standard text display. Therefore, a multimedia application is the one that uses and includes more than one of these media

in a cohesive manner. A multimedia language is "a set of software tools for creating multimedia applications" [73]. All multimedia languages present the developer with a set of software tools to aid in the development process. Although they all aid in manipulating similar presentation media, the functionality of these software tools may vary greatly from language to language. As of yet, no standard development environment exists. Table 4-5 shows three metrics for the multimedia application. The Number of External Interactions (NEI) measure is to evaluate the interaction triggered by the users. The Percentage of Spatial Relations (PSR) and the Percentage of Temporal Relations (PTR) measures are used to quantify the spatial relations and the temporal relations respectively.

Metrics	Definition
NEI	Number of external interactions triggered by users
PSR	The percentage of spatial constructs in the total constructs in the program
PTR	The percentage of temporal constructs in the total constructs in the program

Table 4-5 Multimedia Specific Metrics

4.3.6 Feature Oriented Metrics

Features are an effective media of communication between users and developers. On the one hand, users focus on the problem domain to present their maintenance needs such as adding a new function, where the system's features are the primary concerns. A feature is a unique identifiable characteristic of an application domain in the view of users or developers. It is represented by a single term or term pair. There are three kinds of features [62]:

- Capability or functional features express the services or the way users may interact with a product;

- Interface features express the product's conformance to a standard or a subsystem;

- Parameter features express enumerable, listable environment or non-functional properties.

A very basic principle in object-oriented software engineering states that a class should implement one single feature of the application domain [59]. Some violations of this principle can be detected by using these assumptions: (i) a class that implements more than one feature has probably low cohesion measurements, since these features can be implemented separately, (ii) a class that by itself does not implement one feature (the concept is distributed among many classes) could be tightly coupled to other classes. Therefore, by collecting cohesion and coupling values of an object-oriented legacy system, possible violations of the principles 'one class – one feature' [59] can be found. Having such feature-oriented classes with high cohesion and low coupling, software can be equipped with high maintainability, reusability and accordingly easily evolve. The feature-oriented measures are used to quantify such high cohesion and low coupling so that the one can determine how to perform transformations to obtain the feature oriented class.

The process to obtain the feature-oriented class as implies two necessary procedures:

(1) To find the source code that delivers the implementation of a feature

This action is also named as feature location or concept assignment. Feature location is to locate a particular feature in the most relevant code, understand it and make the change to minimise unwanted side effects. Through feature location, the relationship between implementation module and a particular feature can be recreated or recovered.

Many researchers have studied dynamical and static approaches [22, 32, 38, 122, 123, 131] which suggest different way to locating features in their implementation modules. In order to generate fine-grained components, the test case based location techniques are suggested to use, such as [135, 136]. A program slicing technique integrating backward slicing and forward slicing would be used to slice a fine-grained executable module that serves a particular feature. Through program slicing technique [119], the irrespective pieces of source code and variables can be sliced off and only the related code blocks are left. The location relationship on the fine grain implementation can be described by a cross reference table.

The first procedure is not included in the scope of the thesis. It is assumed that the feature location has been done through the existing techniques. The program transformations to be applied will be focused on the second procedure.

(2) To aggregate the located source code as a software module with high cohesion, such as a class or a component

The aim of this step is to aggregate implementation modules which are involved in a particular feature. After identifying the source code that is involved in the implementation of a particular feature, the implementation modules are aggregated. In addition, the interrelationship between features and implementation modules as well as interaction between features can be discovered and used for the design recovery process.

In order to formalise the mapping relationship between a feature and its implementation module, the follow notation is given:

FE: a feature;

FR $_{ij}$: relationship between FE_i and FE_j in a feature model which describes the relationship between features;

FIM $_j$ (V,F): the feature implementation module which is the located implementation modules for a feature FE_j;

$V_j = \{v_1, v_2, \cdots, v_n\}$ is a set of data used in FIM_j;

$F_j = \{f_1, f_2, \cdots, f_n\}$ is a set of functions implemented in FIM $_j$, where $f_i, i = 1, \cdots, n$ represents a function. In object-oriented system, the reference of an attribute of an object also can be regarded as a calling function;

Com: a feature-oriented module in the new system which fulfils the feature oriented cohesion target;

Conn: the connection between components;

The mapping relationship between FIM and FE is $FIM(V, F)$ = locate (FE), where locate represents the process that FE is located in source code as $FIM(V, F)$.

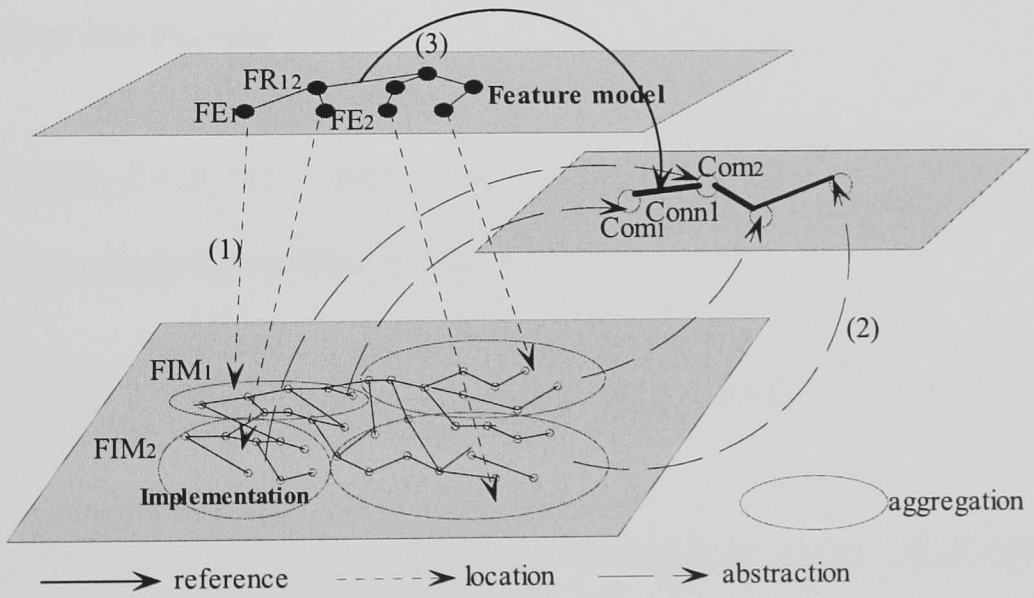

Figure 4-1 Feature-Source Code Mapping Relationship Diagram

The interactions of features can be reflected as the share part of their FIM. Meanwhile, the relationship between features in the feature model is a reference for constructing the

connections among components in the design model. The mapping relationships between the different abstraction modules are depicted in Figure 4-1.

Metha [77] proposed four interactions between FIM to construct feature-oriented components.

- Shared Stateless Functions (SS): A stateless function can be shared between two feature implementation modules (FIM).
- Shared State-Full Functions (SSF): A state-full function can be shared between two FIMs.
- Dependent Data (DD): An FIM may be dependent on the data accessed by another FIM.
- Dependent Function (DF): An FIM may be dependent on a function that is part of another FIM.

To construct a feature oriented module based on the feature interaction, the following rules are taken into account [77].

Rule 4-1 If $\exists SS, f \in F_1 \cap F_2$, then f is not encapsulated within Com_1 and Com_2, but its state is accessed via public interface.

Rule 4-2 If $\exists SSF, f \in F_1 \cap F_2$, then f is encapsulated within Com_1 and Com_2.

Rule 4-3 If $\exists DD, v \in V_1 \cap V_2$, then v leading to message communication between Com_1 and Com_2.

Rule 4-4 If $\exists DF, f \in F_1 \cap F_2$ then f is encapsulated within Com_1 and Com_2 and

gives a clue to specify the message communication of the two components.

Rule 4-5 If $F_1 \bigcap F_2 = \Phi \wedge V_1 \bigcap V_2 = \Phi$, FIM_1 is encapsulated within Com_1 and FIM_2 is encapsulated within Com_2.

Feature oriented metrics shown in Table 4-6 are used to measure the cohesion and the coupling of the features. A high cohesion component for a feature has the high reusability and maintainability.

Metrics	Definition
ANFC	Average number of features implemented in a class
ANSDF	Average number of shared data module between two features
ANSFF	Average number of shared functions between two features
NCIF	Number of classes implementing a feature
OSC	Overlap statements among the classes

Table 4-6 Feature Oriented Metrics

4.4 Reengineering Target Definition and Modelling

Over the past decade, goal models have been used in computer science in order to represent software requirements, business targets and design qualities. Such models extend traditional Artificial Intelligent (AI) planning techniques for representing goals by allowing for partially defined and possibly inconsistent goals [45]. The Non-Functional Requirements (NFR) approach [83, 84] is based on the notion of soft-goals rather than (hard) goals. By referring the structure of the soft-goal model, a Target Model (TM) is proposed. Using 'target' instead of 'soft-goal' because a target is referred to not only a soft-goal which is referred to improve non-functional quality, but also a task of software reengineering, which is needed in most of software reengineering activities. For example, abstraction is not a soft-goal but a very useful means for understanding legacy code and adding functions to the existing system. The

specific definitions related to target are given in Definition 3-1 and Definition 3-2.

Referring the concept of goal-driven techniques [84], a target is satisficied rather than achieved. Target satisficed degree is introduced to express that target factors as sub-targets are expected to achieve the parent target within acceptable limits, rather than absolutely. Target non-satisficed is based on the notion that the target is never totally achieved or not achieved. Targets can be related to their target factors in terms of relations such as AND and OR [84]. The meaning of these relations has as follows:

- $\mathrm{AND}(G, G_1, G_2, \cdots, G_n)$ — target G is fulfilled when all of $(G, G_1, G_2, \cdots, G_n)$ are fulfilled and there is no negative evidence against it.

- $\mathrm{OR}(G, G_1, G_2, \cdots, G_n)$ — target G is fulfilled when one of $(G, G_1, G_2, \cdots, G_n)$ is fulfilled and there is no negative evidence against it.

Given a constraint as a target factor for a transformation problem, one can look up the target interdependency graph for that constraint and examine how it relates to other target and what are additional transformations that affect the desired target positively or negatively. The target interdependency graph is called Target Model (TM) containing the target factors which interrelate to each other by the above relations and are constructed within the model.

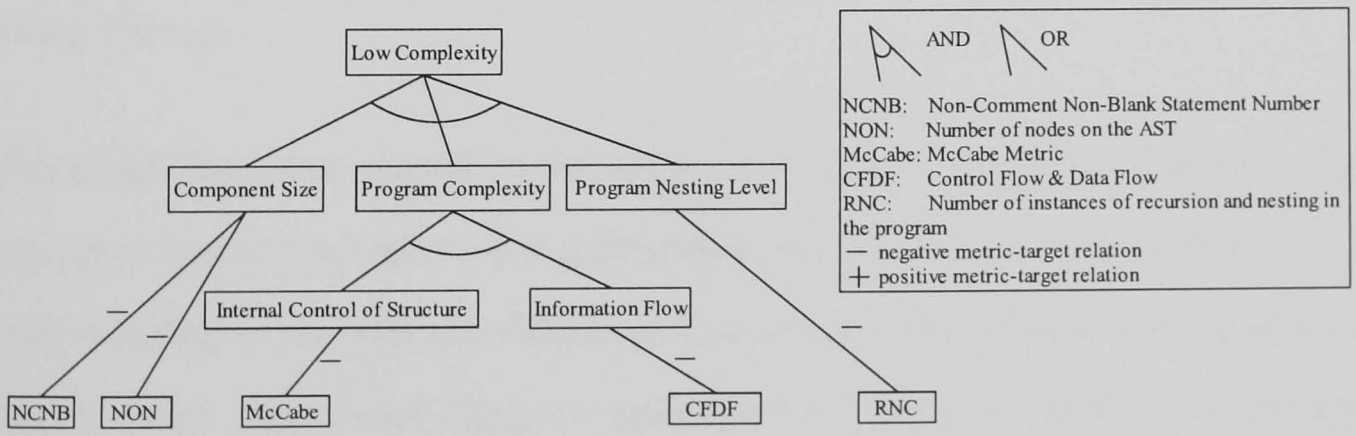

Figure 4-2 Target Model of 'Low Complexity'

Figure 4-2 shows an example of the target model which represents the 'low complexity' target. The root node on the tree is the pursued target. 'Component Size', 'Program Complexity' and 'Program Nesting Level' are the target factors to the top target. 'Program Complexity' has its own target factors, such as 'Internal Control of Structure' and 'Information Flow'. The leaf nodes are selected metrics attached to the related target for measuring the target satisficed degree. The usage of metrics in the TM will be stated in Section 4.5.

4.5 Measurement of Target Using Software Metrics

In addition to being equipped by the target model, to realise the target-driven transformation step prediction needs more means which can measure the effects of transformation and provide the quantitive information to guide the transformation process. In the context of this issue, a number of software metrics and features that are related to the reengineering have been examined in the previous section.

For the targets modelled in the interdependency graph, a set of metrics are selected to compute the corresponding source code features, appearing as leaves in TM. The metrics indirectly reflect the satisficing of the direct or indirect parents in the TM. Furthermore, each program transformation is associated with a collection of features it affects and consequently with the magnitude of change in the corresponding targets being affected.

The attachment of the metrics to the target model depends on what the target is. User can select the relevant metrics for an identified target. The selection relies on the user's understanding of the problem domain and the usage of the metrics. This is at much higher level and easier than to understand a massive number of program transformations. The composition of the metrics can be justified by a domain expert.

The relationship between targets and metrics is not necessarily one-to-one, that is, a single target may be associated to more than one metrics. For example, as shown in Figure 4-2, 'Component Size' can be measured by the number of non-comment non-blank statements or the number of nodes on the AST.

It is allowed that a metrics contributes multiple targets. However, this is not suggested because too many interlaced relationships could negatively affect the analysis results. A metrics herein is regarded to measure a single target factor. To simplify the algorithm, the given metrics are for the leaf nodes of TM merely. Therefore, only leaf nodes of TM can be measured by the metrics. The nodes over leaf nodes do not have relations with any metrics.

To specify the quantitative relation between metric and target, there are two kinds of relations, i.e. positive relation + and negative relation –.

- $+(m,G)$ — metric m contributes positively to the fulfilment of target G.

- $-(m,G)$ — metric m contributes negatively to the fulfilment of target G.

If the relation between metric and target is modelled as a function $f(x) = y$, where x is the value of a metrics and y is the satisficed degree of target, then the two kinds of satisficing relations can be described as follows.

Positive metric-target relation '+': a metrics positively contributes its parent if and only if the relation function is increasing, i.e., $f(b) > f(a)$ for all $b > a$. This kind metrics provide more benefit by increasing their values. The metric is called to be positive to the target.

Negative metric-target relation '–': a metrics negatively contributes its parent if and

only if the relation function is decreasing, i.e., $f(b) < f(a)$ for all $b > a$. This kind metrics provide more benefit by decreasing their values. The metric is called to be negative to the target.

For instance, the 'Low Complexity' is negatively contributed by the 'Lines of Code' metric and the program complexity.

Formula 4-1 gives the formula developed by the author to compute the measurement related to a target modelled in the TM. The formula is generated based on the three points. (1) The AND/OR relations in the TM. (2) The positive and the negative effects of metrics on a target. (3) The normalisation of the metrics.

Formula 4-1 Given a target τ which is modelled as the target model Γ, μ_τ is denoted as the computation of the target, then

$$\mu_\tau = \begin{cases} \sum_{i=1}^{p} c_i\omega_i - \sum_{i=n+1}^{q} c_i\omega_i & \text{if } AND(\tau,\tau_1,\cdots,\tau_p,\tau_{p+1},\cdots,\tau_q) \\ & \wedge m_1,\cdots,m_p \text{ is positive} \wedge m_{p+1},\cdots,m_q \text{ is negative} \\ -\min(\omega_1,\cdots,\omega_p) & \text{if } OR(\tau,\tau_1,\cdots,\tau_p) \\ & \wedge \min(m_1,\cdots,m_p) \text{ is negative} \\ +\min(\omega_1,\cdots,\omega_p) & \text{if } OR(\tau,\tau_1,\cdots,\tau_p) \\ & \wedge \min(m_1,\cdots,m_p) \text{ is positive} \end{cases}$$

where ω_i is the computation function of τ_i, if τ_i is not a leaf node of the target model, or $\omega_i = f_j(m_i)$, $f_j(m_i)$ is the normalisation method to normalise the value of the metric computation corresponding to the τ_i if τ_i is leaf node of the target model and represents a metric. In addition, some sub-targets are more important than others and in this case goal weights are determined by the users and are added as a coefficient c_i.

The above definition of the computation of a target is recursive because the computation of the sub-targets can also be obtained by the formula. The sub-targets' general scores contribute the super-target. The computation formula can be used to measure the impact of program transformations on targets. The usage will be explored in Chapter 6.

For example, for the TM shown in Figure 4-2, the computation of the target 'Low Complexity' which is the root target in the model can be calculated as follows.

$$\mu_{LowComplexity} = \min(f(NCNB) + f(NON)) + (f(McCabe) + f(CFDF)) + f(RNC)$$

The purpose to use normalisation function is to normalise the values in different ranges into the range [0..1]. The normalisation function is defined according to the scenario of problem. In Chapter 6, the normalisation function will be defined in more detail for constructing transformation impact function.

4.6 Summary

This chapter gives the method to modelling the top level of MOTMET introduced in Chapter 3 including the representation of target and the correlated measurements. The following points can be summarised from the chapter.

- The definition of target is given and particularly refers to the objective of reengineering activity.
- Six categories of reengineering-intensive software metrics are presented. These metrics are grouped and chosen in that their capabilities to measure reengineering activity.

- Correlated with the metrics, reengineering target can be modelled by the target model by referring the existing goal modelling technique.

- In order to evaluate the target satisficed degree, a formula is proposed for this computation.

Chapter 5
Extension of Wide Spectrum Language and Transformation Bank

Objectives

- To extend *Meta*WSL with the target-driven features
- To extend WSL with the object oriented features based on the hierarchy of WSL
- To extend WSL into the specific domain, using multimedia as a paradigm
- To extend the program transformations based on the extended WSL
- To present the meta-model and the catalogue of the program transformations contained in the transformation bank

5.1 Introduction

The WSL language and transformation theory was created for software maintenance, reverse engineering and migration. However, the application of WSL is limited in unstructured program and procedural program. In this chapter, the application and extension of WSL and the transformation bank is discussed. Object-oriented extension

will be performed from both of the syntax and the semantics aspects. By using WSL as an intermediate language to extend the application of transformation into the specific domains is another key in this chapter. The multimedia domain will be explored for the application of WSL's extension. Due to the extension, WSL can be extended by adding more features for the new application in those domains. Furthermore, the extension of transformations based on the extended WSL will be presented, correspondingly to extend the transformation bank. In order to support the transformation prediction, a method to manage the transformation bank is also described.

5.2 WSL Extension Approach

In Section 2.7.2 is introduced the extension of WSL kernel language by adding three levels which cover the main syntax constructs of procedure HLL. The stepwise addition of language levels enriches the flexibility of WSL and brings the high possibility to extend WSL. The new language levels are built up with the language at each level being defined in terms of the previous level. The kernel language forms the foundation for all the others. Each new language level automatically inherits the transformation proved at the previous level; these form the basis of a new transformation catalogue. Transformations of each new language construct are proved by appealing to the definitional transformation of the construct and carrying out the actual manipulation in the previous language level [125].

Due to the capability and flexibility of WSL, the extension of WSL can be manipulated consistently based on the existing hierarchy of the language, especially on the semantic aspect which is the fundamental perspective for implementing and proving the semantic preserving transformations. Figure 5-1 gives a paradigm of the WSL extension which aims to add object-oriented features to generate a new level of WSL

and augment the application of WSL into some advanced domains by adding their domain features.

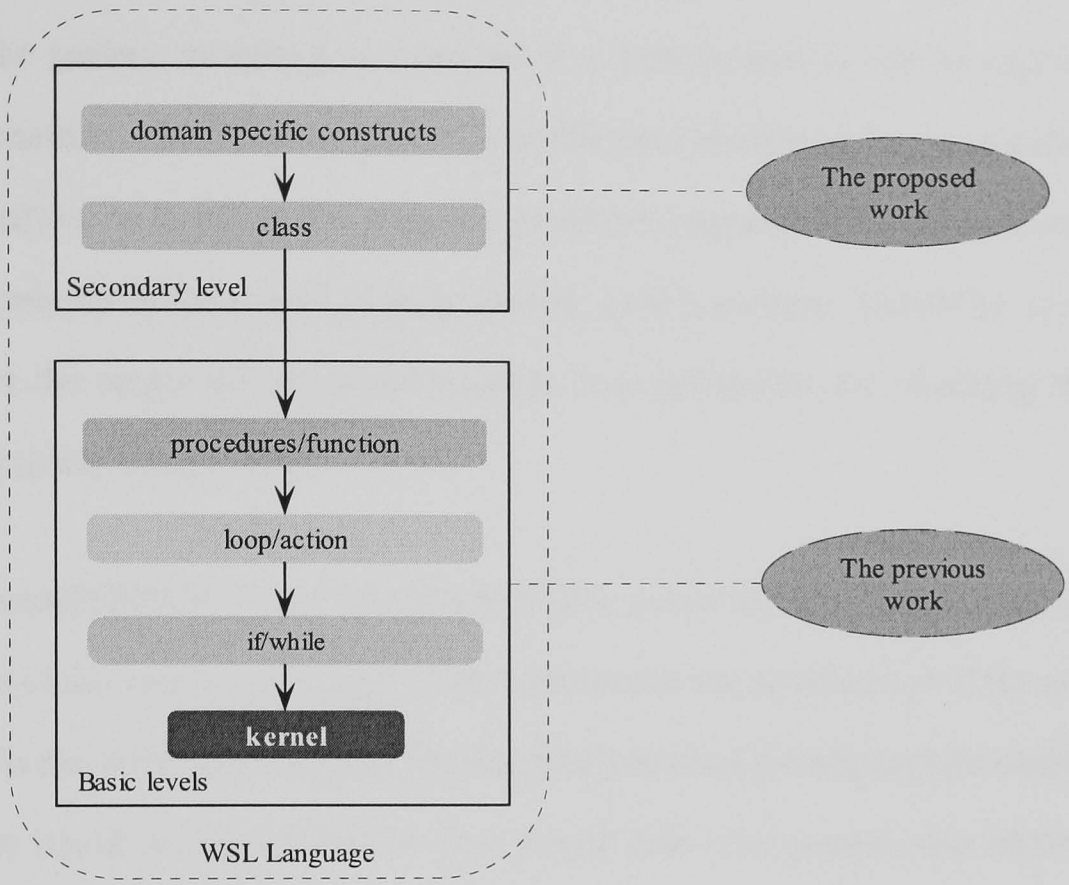

Figure 5-1 Extension Model of WSL

5.3 WSL Extension for Supporting Target Driven Reengineering

In the FermaT transformation system, the program transformations are carried out in WSL and the transformations themselves are written in an extension of WSL called *Meta*WSL which was specifically designed to be a domain-specific language for writing program transformations. As a result, FermaT is capable of transforming its own source code via meta-transformations [118].

In order to automatically predict program transformation steps driven by target, it is necessary to test the current status of source code to determine the applicable

transformations. This needs to parse the source code and extract a list of constructs for matching the application condition of transformations. In the existing toolset, it is implemented by two important constructs, i.e., **ifmatch** and **fill**. The two constructs support the pattern matching to examine if a transformation can be applied for the current construct. However, the **ifmatch** or **fill** must be followed a given pattern for the transformation. In terms of the proposed prediction approach, the identification of the existing patterns in the source code is needed. In this section, *Meta*WSL are extended to support the target driven transformation step prediction for checking the pattern contained in the source code.

@Get_Trans(POSN,SC): this function is used to detect the appropriate transformations for a node of the source code AST. POSN represents the position list of the node on the AST. SC is the WSL source code. The function returns a list of program transformation candidates which are applicable on the current item. The pseudocode of the function will be given as Algorithm 6-3 in Chapter 6.

For example, given the **if** statement as follows.

```
if (m = 1) then
    p    : = number[i];
    line : = ((line++", ") ++ p);
fi;
```

The result returned from @Get_Trans(POSN,SC) is a list containing the applicable transformations <absorb, add_comment, add_skip, insert_assert, simplify, simplify_all_expressions, replace and simply, replace_all_values_optimally>.

The function will be used for constructing the transformation process model for the target driven prediction.

5.4 Extension of WSL with Object Oriented Features

Most of the current applications of using WSL are on the procedural programming. There have not been the concepts of object orientation obviously. However, the theoretical and implementation of WSL provides a foundation for extending with these object oriented features.

There are two aspects taken into account for this extension.

- Semantic aspect: Theorem 2-1 about recursion can be used to define the important object oriented concepts. In the following sections, the relevant definitions and theorems will be given based on the recursion.

- Syntactical aspect: The existing relevant constructs of WSL are at the third level. The constructs VAR...ENDVAR and STRUCTURE in WSL can be referred for the definition of class. Upon the third level, the fourth level will be introduced which specifies the object-oriented constructs and relations, such as inheritance, instantiation, reference and polymorph. The Backus Naur Form (BNF) of object-oriented extension of WSL is given in Appendix A.

5.4.1 Definition of Class and Object in WSL

Inheritance is a very important mechanism to support reusing code in object-oriented language, which is for expressing similarity among classes, simplifying the definitions of classes similar to one(s) previously defined. Cook [27] defined inheritance as a mechanism for differential or incremental programming. He proposed a simple form of inheritance as illustrated in Figure 5-2, where P is the original function, M is the modification and the arrows represent invocation. In [69], the work was modified for

enhancing reusability and maintainability in object-oriented language.

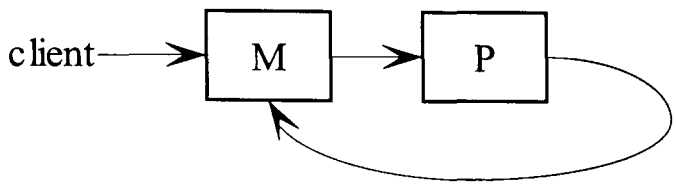

Figure 5-2 Inheritance Model

This construction represents the essence of inheritance: it is a mechanism for deriving modified versions of recursive definitions. Changes derived from incremental programming can be made either to the input passed to the original module or the output it returns, but the way in which the original works cannot be changed. However, there is one way in which the inheritance is in the treatment of self-reference or recursion in the original structure. As shown in Figure 5-2, the modification affects external clients of the function – it modifies the function's recursive calls and the self-reference in the original function is changed to refer to the modification.

To develop the informal account of inheritance into a formal model of inheritance in object-oriented WSL, the fixed point semantics of recursive definitions which has been used in the kernel language semantics can be referenced for this purpose. The central theorem of fixed point semantics of recursive definitions and the relevant definitions [117] in WSL may be stated as follows.

Definition 5-1 The 'flat' order on states $s \sqsubseteq t$ is defined as true when $s = \perp$ or $s = t$.

Definition 5-2 If $f, g \in F_H(V,W)$ are state transformations then $f_1 \sqsubseteq f_2$ iff:

$$\forall s \in D_H(V).(\forall t_1 \in f_1(s).\exists t_2 \in f_2(s).t_1 \sqsubseteq t_2 \wedge \forall t_2 \in f_2(s).\exists t_1 \in f_1(s).t_1 \sqsubseteq t_2)$$

An equivalent formulation is: $f_1 \sqsubseteq f_2$ *iff* $\forall s \in D_H(V).(\perp \in f_1(s) \vee f_1(s) = f_2(s))$.

Definition 5-3 (Monotonic) A function F on state transformations is monotonic if $\forall f \in F_H(V,V).f \sqsubseteq F(f)$.

Definition 5-4 (Directed Set) A directed set F is such that for every $f_1, f_2 \in F$ there exists $g \in F$ such that $f_1 \sqsubseteq g$ and $f_2 \sqsubseteq g$.

Definition 5-5 (Continuous) A monotonic function F on state transformation stratifies $F(\coprod F) = \coprod \{F(f) \mid f \in F\}$ for every directed set $F \subseteq F_H(V,V)$ then F is continuous.

The procedures and functions in WSL are continuous [117].

Theorem 5-1 (Fixed Point) if a function $F \in F_H(V,V)$ is continuous, then there is a state transformation $(\mu.F) \in F_H(V,V)$ such that $(\mu.F) = F((\mu.F))$. This $(\mu.F)$ is called the least fixed point of F, written $fix(F)$. It is given by $\coprod_n F^n(\perp)$.

Throughoutly, functions like F will be called *generators*. The fixed point semantics was used to describe the behaviour of objects with mutually recursive methods in [27]. Referred the work in [27] and the theory of WSL, an object-oriented denotational semantics of WSL is stated as follows.

Definition 5-6 A structure containing a set of variables and procedures or functions defined in WSL, denoted by R: $\begin{bmatrix} x_1 \\ \vdots \\ x_n \\ M_1 \\ \vdots \\ M_m \end{bmatrix}$, with variable $x_i, i = 1 \cdots n$, called fields and

$M_j, j = 1 \cdots m$, called methods which represent procedures or functions.

Definition 5-7 A record is a set of states $B \subseteq D_H(V)$ and denoted by $\begin{bmatrix} x_1 \mapsto v_1 \\ \cdots \\ x_n \mapsto v_n \end{bmatrix}$ with variables x_i called labels and values v_i.

The relevant definition of a class can be defined as follows.

Definition 5-8 Let H be a set of values. A structure R is a class that $(\mu.R) = \coprod_{DF\ n<\omega} R^u(\Omega)$, where $R^n(\Omega)$ is the 'nth truncation' of $(\mu.R)$, i.e., $R^n(\Omega) = H_n \cup \mathbf{P}(R_{H_n} \setminus \{\perp\})$, $\mathbf{P}$ is a power set. The $\coprod$ operation collects all this information together to form $(\mu.R)$. The fixed point of $(\mu.R)$ as a record is an object of R. Class R is modelled as a generator.

Syntactically, a class definition is a named construct, optionally with generic parameters. In this construct shown in Figure 5-3, the constituents of the class are defined and related. The main constituents are: fields and methods.

Class	ClassName [generic parameters]: inherited classes
	state schema
	initial state schema
	operation schema

Figure 5-3 Class Construct

Here, the state schema is the field declaration. The initial state schema is the constructor of the class. Operation schema actually is the method declaration.

```
Class Point(a,b)
        field x
        field y
        method x = a
        method y = b
        method distance (p) =
            sqrt(square(self.x-p.x) + sequare(self.y-p.y))
```

Figure 5-4 The Class 'Point'

For example, class Point in Figure 5-4 is modelled as a generator MakeGenPoint(a,b), defined in Figure 5-5. MakeGenPoint takes the coordinates of the new point and returns a generator, whose fixed point is a 'point'.

As discussed in the last section, the fixed point semantics of class also defines the semantics of instantiation, i.e., an object P = $fix(\mu.Point)$, where Point is a generator of an object. The keyword self embedded in a class represents the self-reference of the class. Its definition in WSL will be given later. A point (3, 4) is created as shown in Figure 5-6 .

```
MakeGenPoint(a,b) =μ.
    [ x ↦ a,
      y ↦ b,
      distance ↦
            sqrt(square (self.x-p.x)+ square (self.y-p.y))]
```

Figure 5-5 The Generator Associated with 'Point'

```
p  =  fix(MakeGenPoint(3,4))
   =  {
         x ↦ 3,
         y ↦ 4,
         distance (5,6) ↦ 2,
      }
```

Figure 5-6 A Point at Location (3, 4)

In other words, the correspondence between object-oriented terminology and semantics

of WSL can be listed as: objects are modelled as record value whose fields represent methods; the values may refer recursively to the whole record.

If *R* is the name of a class, the identifier *R* semantically also denotes the set of identities of possible objects of class *R*. (In particular cases, it will be clear from the context whether an identifier is being used as a name of a class or to denote the set of identities of objects of that class.) Informally, we do not distinguish between an object and its identity, i.e. if we refer to some *object* of *R*, it is with the understanding that we are in fact referring to an object whose identity is in *R*. Because object identities uniquely identify objects, no ambiguity arises.

Syntactically, in WSL, the object instantiation is defined as follows.

<class instance creation expression> ::= **new** <class > (<argument list>?)

<argument list> ::= <expression> | <argument list> , <expression>

<class > ::= <class name>

For example, to instantiate an object from the class **Point**, we can have:

```
p = new Point (3, 4);
```

Figure 5-7 An Object of Class Point

The class construction is the essential extension to WSL with Object Oriented Features (OOF), which groups the definition of schema including the state schema and the initial schema and the definitions of its associated operations. A class is a template for objects of that class: for each such object, its states are instances of the state schema of the class and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definition of its

class. On the above definitions, the fixed point semantics of class also defines the semantics of instantiation.

WSL is a weak typed system. Although, the type system discussion is outside of the scope of this thesis, as for the object-orientation, type cannot be neglected completely. In object-oriented program, type is a term including two aspects, primitive type and reference type. In the extension of WSL, reference type will be introduced for the inheritance and instance relations.

After defining the essential constructions and giving the basic theorem, we continue to discuss the inheritance based on WSL semantics.

5.4.2 Inheritance

Inheritance is modelled as an operation on generators that yields a new generator. There are three aspects to this process: (1) addition or replacement of methods, (2) the redirection of self-reference in the original generator to refer to the modified methods and (3) the binding of super in the modification to refer to the original methods.

For example, the **Point** class may be inherited in defining a class of circles. Circles have a different notion of distance from the origin. This definition gives only the differences between circles and points:

```
Class   Circle(a,b,r) : Point(a,b)
            field radius
            method radius = r
            method distFromOrig
                = max(super.distFromOrig – self.radius, 0)
```

Figure 5-8 The Class Circle Inherited from the Class Point

The formal interpretation of inheritance presented above is formalised using generators. The essential observation is that the manipulation of self-reference can be modelled as an operation on generators.

Definition 5-9 Inheritance is the derivation of a new generator from one or more existing generators, in such a way that the formal parameter of the derived generator (representing self) is passed to all of the inherited generators.

The new generator inherits from the original generators. The original generators are called ***parents***; the derived generator is called the ***child***. When there is no chance of ambiguity, the corresponding fixed points may also be called the parent and child.

Since generators are closely related to self-referential definitions, the effect of inheritance can be understood as an operation on definitions. In this context, inheritance corresponds to textually embedding an existing definition inside a new definition, when using the syntactic convention of representing self-reference by the keyword **self**. Since the same identifier is used to represent self-reference in the inherited definition and the definition in which it is embedded, self-reference is shared between them.

The modifications effected during class inheritance are naturally expressed as a record of methods to be combined with the inherited methods. The new methods M and the original methods O are combined into a new record $M \oplus O$ such that any method defined in M replaces the corresponding method in O.

The modifications, however, are also defined in terms of the original methods (via **super**). In addition, the modifications refer to the resulting structure (via **self**). Thus, a modification is expressed as a function of two arguments, one representing **self** and the

other representing **super**, which returns a record of locally defined methods.

5.4.3 Reference

In Object-Oriented WSL, a class consists of declarations and imperatives. The declarations define the attributes possessed by instances of the class. The imperatives are procedure or function in WSL that is evaluated when an instance is created. They provide the primary behaviour of the class. Thus, a class may be viewed as a combination of the block and procedure concepts.

Attributes may be accessed from outside the instance by a reference to the instance. Imperative can be access by method invocation.

Class T : A
{
field x_i;
method $m_j(In\ pin_{jk} : T_k, Out\ pout_{jl} : T_l^{'})$
$\{A_j\}$
}

This statement shown above is the class building declaration. It defines a class named T, which has attributes, i.e. data fields x_i, $i \in 1...n$ and methods m_j, $j \in 1...r$. pin_{jk} stands for the input parameters of method m_j and $pout_{jl}$ stands for the output parameters of method m_j. The input parameter passing convention is *call by value* and the output parameter passing convention is call by value_return of WSL procedure. A_j is the methods body of method m_j.

Let object t is an instantiation of Class T, t = **new** T; the attribute reference is

represented as $t.x_i$; the method invocation is represented as $t.m\ (In\ e_k, Out\ y_1)$ which invokes the method m in object t.

To the specification of the class instantiation and the object reference can be written in the program context as follows.

begin object
p = **new** P( );
$v_1 = p.x_1; \cdots; v_n = p.x_n$;
$p.F_1$,
$\cdots$,
$p.F_m$
where
class P : S
field $x_1, \cdots, x_n$
method F_1(P **var** r),
$\cdots$
method F_m(P **var** r),
method F_q(P **var** r)
self. F_1;
end
end

5.5 Utilisation of Domain Features for WSL Extension

An essential task of WSL and its transformation theory is to analyse and understand source code through program transformation. The features of source code which is translated to WSL are taken into account in the development and research on WSL, such as the error handling construction of assembler program. However, the domain features have not been used as a means for applying program transformation. In many cases, domain features are important knowledge for any analysis process on source code, such as program comprehension, reverse engineering, maintenance and so on.

An aim of the thesis is to utilise domain features within the program transformation.

The application from a specific domain is chosen for the proposed research, i.e., multimedia application. The reason to choose them is driven by their particular data structures and operations on the data. For example, multimedia data can be divided as static data and dynamic data, depending on whether or not the data change over time.

In the proposed approach, WSL is extended with the capability of dealing with and analysing such applications. The extension focuses on the domain features of the areas. As discussed above, the data and its control in the domains are the triggers for the extension.

However, a challenge of the extension to the advanced domains is the definition of semantics in the domains, i.e., the behaviour of their applications. The essential of the transformation based on WSL is to alter the presentation of program without changing the behaviour of it. The definitions of program behaviour alter over different domains. On the other hand, the definition of the semantics will determine the application of the program transformations. Therefore, it is important to define the semantics in a domain for the transformation purpose. As a basis of the discussion, the semantics of WSL also need to be modified according to the specific domains.

5.5.1 Domain Features of Multimedia Application

More recently, multimedia has appeared as a strong new force within the field of information technology. The field is at the crossroads of several major industries: computing, telecommunications, publishing, consumer audio-video electronics and television/movie/broadcasting. Multimedia applications have existed at every corner in our life.

The models of multimedia software described in [101], a multimedia application is

constructed from a collection of multimedia objects. The primitive objects are media objects of the same media type. The complex multimedia objects are composed from these primitive objects and in general are of mixed media types. Spatial and temporal composition rules must be taken into consideration.

5.5.1.1 Modelling Structure and Behaviour of Media Data

Structural models for multimedia data provide a framework for the structural representation, interpretation and processing of media data. Other data related to multimedia data units shown in Figure 5-9 are the so-called metadata. The metadata, such as document data, representation of the primary document structure and structure representation of the media, are used to describe the relationship between the multimedia data and manage those data. Documents describe media data by means of a formal language, which can be processed by a computer. Documents play two distinctive roles within multimedia data modelling. First document data are handled in a binary representation by the Database Management System (DBMS), like unprocessed media data. For efficiency reasons, often only a partial interpretation of the internal structure of the documents is performed. Second, document types are based on the same abstraction used for the structural formula of media data.

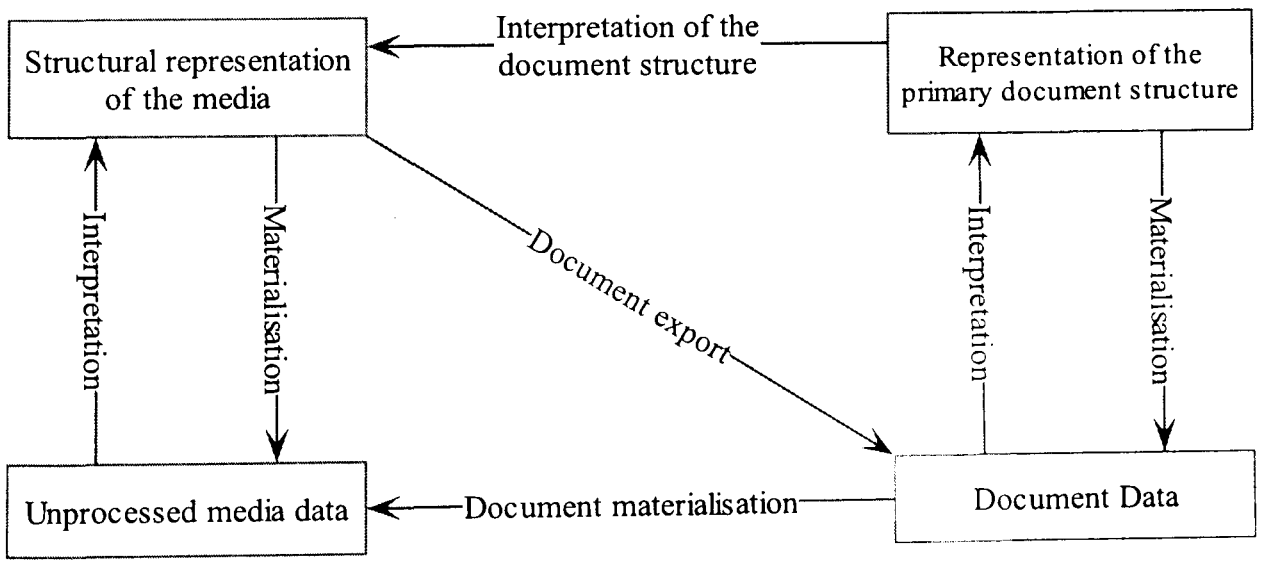

Figure 5-9 Relationships between Media, Documents and Structural Models [101]

5.5.1.2 Data Type Modelling

Different types of multimedia data have different properties and functions, so that it appears intuitive to describe them by different data types. One possible division could be drawn between static and dynamic data, depending on whether or not the data change over time. In turn, the dynamic data can be divided into discrete and continuous data according to how they change over time. Figure 5-10 demonstrates this division.

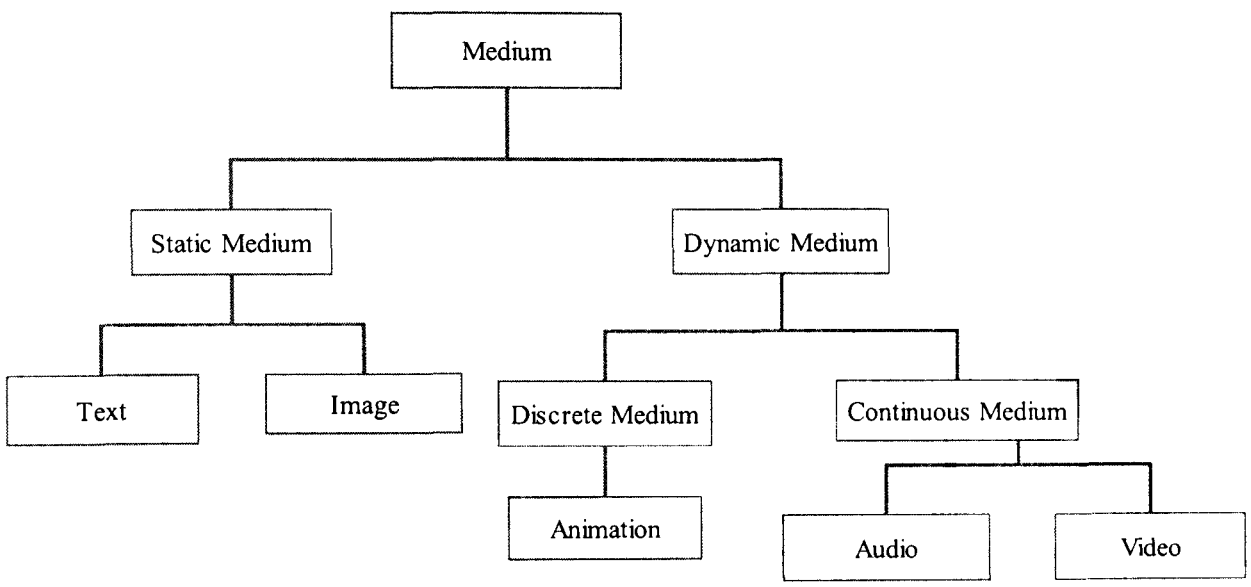

Figure 5-10 Typical Specialisation Hierarchy for Multimedia Data [101]

5.5.2 Extension of WSL with Multimedia Features

Multimedia application involves a variety of individual multimedia objects presented according to a set of specifications. These multimedia objects are transformed (spatially and/or temporally) in order to be presented according to requirements. Multimedia documents are composed, in time and space, of different media, i.e. video, audio, image or text. Three important reasons make the management of multimedia document a complex task: (1) the priori 'unstructured' (or at the best 'semi-structured') characteristic of multimedia document, (2) the organisation of the media content over space, (3) the inherent temporal dimension of media (like audio and video). A

multimedia document can be considered as semi-structured. Indeed, its structure is priori unknown, irregular, without any generic definition, but eliciting and content analysis tools can enable the underlying structure elements to be identified. The events, objects and elements composing the document can thus be identified due to a structure automatically generated by these tools.

In the context of multimedia application, WSL is extended with such multimedia features by defining the media data and the spatiotemporal relations among the data.

5.5.2.1 Media Data Class Declaration

In the object-oriented extension of WSL, the inheritance, instantiation and reference relations are defined semantically and syntactically. The definition will be adopted in the media data definition and declaration.

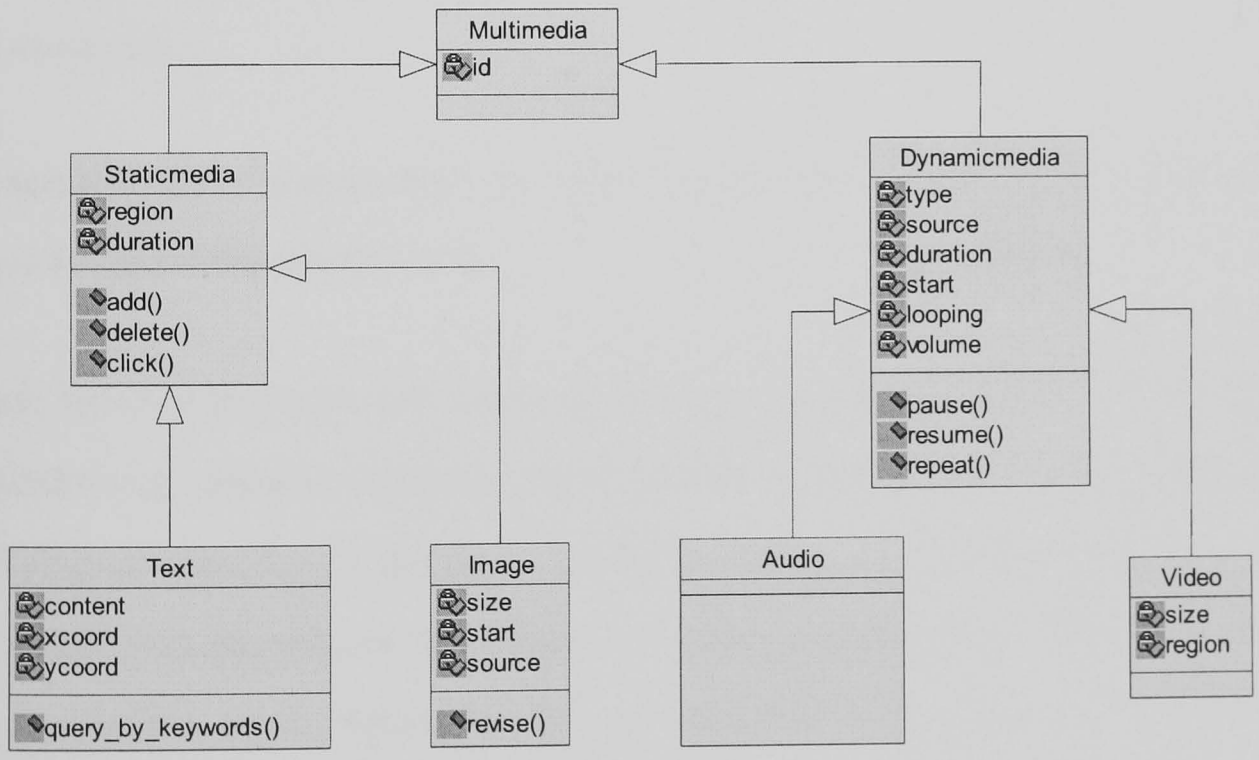

Figure 5-11 Class Diagram of Multimedia Data

As shown in Figure 5-11, there are seven classes defined for the media data. The

superclasses are inherited by the subclasses. The behaviour and properties of each media type is encapsulated into a class. The behaviour is classified as spatial behaviour and temporal behaviour. Particularly, the dynamic data, such as audio and video, have life cycle that is operated as start and stop process. Such features can be modelled as the media's behaviour in the media class. On the other hand, the information can be regarded as metadata of those media data. Below, the further description and the class representations of the media types are listed as follows.

Multimedia: Multimedia is a superclass of all the media data and only contains a property as 'ID' which is inherited by the subclasses, such as *Staticmedia* class and *Dynamicmedia* class.

Staticmedia: *Staticmedia* is a superclass of *Text* class and *Image* class because the two classes represent the static media data. The class includes the properties and actions of the static data.

Dynamicmedia: *Dynamicmedia* is a superclass of *Audio* class and *Video* class which represent particular dynamic data.

Text: Text is information that makes the context of document content intelligible as a structure (e.g. syntactic structure, lexical information, reference, relation, table, etc). Eliciting structure from a textual document enables to identify document items called 'elements'. An element can be a chapter, a section, a paragraph or a sub-paragraph, at any embedded level. Hence, the specific structure of a given text document is hierarchically detailed according to the granularity level (word, sentence, paragraph) of the application.

Image: the segmentation of image can be identified as the shapes and the associated

patterns. The properties like color, texture, shape, key words (through OCR devices) or any regular pattern (objects or regions and their spatial relationship) can be extracted.

Audio: the features of audio can be extracted as amplitude, bandwidth, pitch and duration. As a continuous data, an audio data has its life cycle which can be triggered by start operation and terminated by stop operation. Its life can also be paused and repeated.

Video: video is made of audio-visual information.

By the instantiation relation, the concrete media objects can be declared and instantiated as follows.

t1 := new **TEXT**( )

image1 := new **IMAGE**( );

audio1 := new **AUDIO**( );

video1 := new **VIDEO**( );

5.5.2.2 Media Data Relation Modelling

Continuous media such as audio and video imposes new requirements on document modelling. These requirements are essentially due to the intra-media and inter-media temporal and spatial information. Multimedia document architecture can be defined as an integrated and homogenous collection of data describing and structuring the content and representing their temporal and spatial relationships in a single entity.

There are three ways for considering the structure of a multimedia document: logical structure, spatial structure and temporal structure. Spatial structure usually represents the layout relationships, defines the space used for the presentation of an image. Temporal structure defines temporal dependencies between elements.

Logical relationships: The logical relationships define the hierarchical structure of media, including super and sub relationships. The logical relationships are defined only for the same media type. Therefore, it is an intra-relationship.

Spatial relationships: The spatial relationships have been widely studies as spatial operators for handling spatial objects. Several classifications have been proposed like [36]. Our purpose in the thesis is not to compare them or to define yet another one. The classification are chosen, which proposed by [26], which groups all those spatial elementary operators into a few sets. The list of spatial operators in the WSL extension is: the disjunction, the adjacency, the overlapping and the inclusion. The topological relationships between the objects (disjoint, meet, overlap etc.) shown as Figure 5-12.

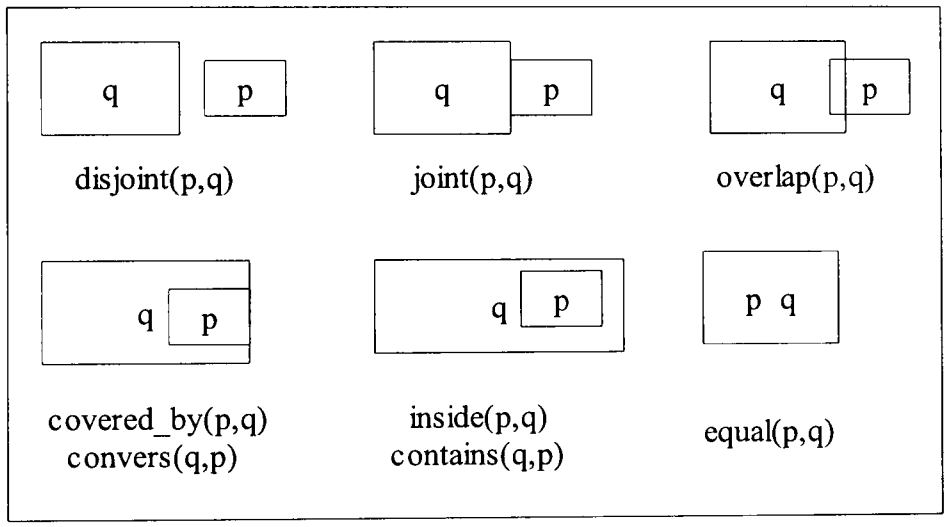

Figure 5-12 Topological Relationships between the Media Objects

Temporal relationships: There are two classifications of time relationships: the time between elements in a same document and the relationships between documents. The first class consists in intra-document relationships, referring to the time among various presentations of content elements. The second one consists in the inter-document relationships, which are the set of relations between different documents (for example the synchronisation of two audio segments during a presentation). The diffusion of media are events and these events are organised using the qualitative relations of

Allen's algebra [5]. In Table 5-1, fourteen temporal relations between media are presented.

Temporal Relation		Symbol	Inverse Temporal Relation	Symbol for Inverse	Pictorial Example
Sequence (X; Y)	X before Y	<	Y after X	>	XXX YYY
	X meets Y	m	Y met_by X	mi	XXXYYY
Parallel (X || Y)	X equals Y	=	Y equals X	=	XXX YYY
	X overlaps Y	o	Y overlapped_by X	oi	XXX YYY
	X during Y	d	Y contains X	di	XXX YYYYYY
	X starts Y	s	Y started_by X	si	XXX YYYYYY
	X finishes Y	f	Y finished_by X	fi	XXX YYYYYY

Table 5-1 Temporal Relations between Media [5]

Definition 5-10 (Sequence Relation) The sequence relation **Seq** is defined as the relation **before** or the relation **meet**, i.e. $x\mathbf{Seq}\ y \in \{(x,y) \mid (x \leq y) \vee (x\ \underline{\boldsymbol{m}}\ y)\}$.

Definition 5-11 (Parallel Relation) **Par** can be presented as one of the following relations, **equal**, **overlaps**, **during**, **starts** and **finishes**, i.e. $x\mathbf{Par}\ y \in \{(x,y) \mid (x = y) \vee (x\ \underline{\boldsymbol{o}}\ y) \vee (x\ \underline{\boldsymbol{d}}\ y) \vee (x\ \underline{\boldsymbol{s}}\ y) \vee (x\ \underline{\boldsymbol{f}}\ y)\}$.

The behaviour of activation is also frequently used in most of multimedia presentation, such as link, trigger and interactivity by user. The defined operators **start** and **finish** satisfy the needs to presenting the behaviour.

5.5.2.3 Syntax Extension of WSL

As aforementioned, the generic structure of multimedia documentation can be modelled as logical, spatial and temporal structure. The three essential relationships also support the composition and synchronisation of media objects. The BNF notation

of multimedia-based extension of WSL is defined for a composite event e as follows.

e ::= e1 Rel e2

e1, e2 ::= text | image | audio | video

Rel ::= <logical relation> | <temporal relation> | <spatial relation> | <boolean_op>

<logical relation> ::= **sub** | **super**

<temporal relation> ::= < | > | = | m | mi | o | oi | d | di | s | si | f | fi

<spatial relation> ::= disjoint | joint | overlap | covers | inside | equal

<boolean_op> ::= **AND** | **OR** | **NOT**

In addition to the above definition, there are two new constructs to define the temporal relations between two media objects, i.e. the sequential construct **SEQ BEGIN...END** and the parallel construct **PAR BEGIN... END**.

SEQ BEGIN S1; S2 **END** $=_{DF}$ Sequence (S1; S2)

PAR BEGIN S1; S2 **END** $=_{DF}$ Parallel (S1|| S2)

The construct '||' is defined by Younger et al. in [128] which also presented its semantics. Therefore, the two new constructs with regard to the temporal relations can be extended semantically by using the existing WSL semantics. The detailed BNF of the WSL extension is given in Appendix A.

5.6 Program Transformation Definition

Over the last three decades, the program transformation method has been proved as a powerful technique for deriving programs from specifications, verifying program properties, specialising programs with regard to their context of use and deriving more efficient program versions from efficient ones. Research on program transformation

has historically focused on semantics-preserving transformations and program refinements – be it in the context of transformational program development or program optimisation. More recently, people have been considering the application of transformation techniques to the case of 'programming in the large' [91].

In the proposed research, the definition of program transformation follows the one given by Ward [117]. Program transformation refers to the process of changing a software system in such a way that it does not alter the external behaviour of the code, i.e., semantics of program, yet improves its internal structure for the specific purposes, such as comprehension and other goals. The semantics here refers the defined WSL semantics. The transformations are applied on the extended WSL source code.

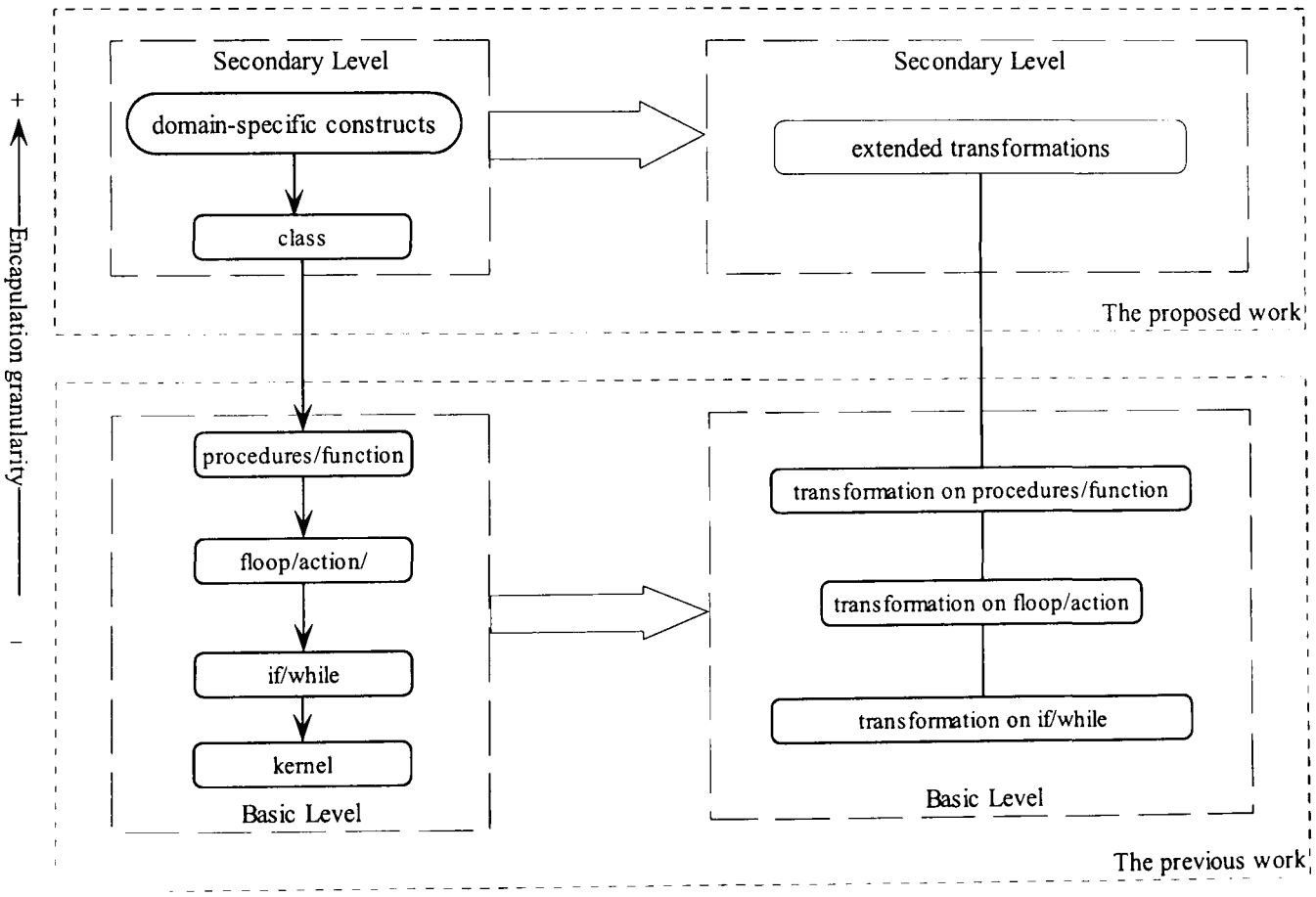

Figure 5-13 Hierarchy of Transformation Mapping to the Levels of WSL

WSL includes a number of construct levels based on the kernel language rising by the encapsulation granularity and allows the extension of the language itself. This feature

of WSL entitles the extension of the program transformations are developed based on the corresponding hierarchical levels shown as Figure 5-13. The transformations can be classified as two groups—basic transformations and secondary transformations.

Basic transformations: the transformations applied on the WSL construct at the basic level.

Secondary transformations: the transformations applied on the constructs at the secondary level of WSL, such as the class and the multimedia specific constructs.

5.6.1 Semantically Equivalent References and Transformation

Intuitively, program transformation should alter the syntax of a program and preserve its behaviour. The transformations existing have been proved to have this capability through Weakest Precondition (WP). However, they are only based on the basic level of WSL language, such as the basic constructs.

Considering the transformations at the secondary level, the semantic equivalency has to be looked at another perspective. A particular set of syntactic and semantic properties of programs were found being easily violated when performing transformations on object-oriented program. Opdyke [86] first discussed the relationship between semantic equivalence and syntax change in the refactoring process. In [86], the semantic equivalence was defined as follows: let the external interface to the program be via the function *main*. If the function main is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same. The definition of semantic equivalence allows change throughout the program, as long as this mapping of input to output values remains the same.

In the development of the transformations at the secondary level of WSL language, the

concept of semantic equivalence adopts the definition used by Opdyke [86]. This allows for several important changes that do not affect equivalence:

- Expressions can be simplified and dead code can be removed. Variables, functions and classes can be removed if they are unreferenced.

- Similarly, variables, functions and classes can be added if they are unreferenced.

- References to a variable and function defined in one class can be replaced by references to an equivalent variable or function defined in another class. One implication of this is that locally defined members can be replaced by inherited members (and vice-versa) if the member declarations are equivalent.

Based on the discussion, the secondary transformations to be developed are allowed to change the internal properties of a program but preserve its external behaviour. The extended transformations developed in the transformation bank are listed briefly in the following section.

5.7 Extension of Transformations

The existing transformations developed in FermaT are based on the WSL language levels rising from the kernel language to the procedure/function level. Most of the transformations work on the basic constructs of WSL, such as WHILE, IF, VAR and so on. Those transformations which have been proved formally alter the syntax of WSL but preserve the semantics of the program. They have been applied in practical projects as well as academic experiments and proved the efficiency.

Nevertheless, as WSL and its application domains are extended, the transformation

bank is expanded for the needs of the extension. In subsequent sections, the transformation extension based the extended WSL which has the object-oriented features or the domain features will be discussed. The transformations called *secondary transformations* in the proposed transformation bank. They are not the composite of the basic transformations, but the ones based on the WSL language extension. Referring [104], the object-oriented transformations adopted are movement transformations, encapsulation transformation and wrapper transformation. The multimedia oriented transformations are proposed based on the extended multimedia constructs in WSL.

5.7.1 Transformations on Object-Oriented Constructs

5.7.1.1 Movement Transformation

The transformation aims to move parts of an existing class to a component class and to set up a delegating relationship from the existing class to its component. This transformation needs three parameters: the name of the existing class (oldClass), the name of the new class to be created (newClass) and the name of the method or field to be moved.

For the applicability of the transformation, the condition function is needed to evaluate precondition in the source code features: (1) the oldClass must exist, (2) the name of the newClass must not be used and (3) the methods or the fields to be moved must belong to the oldClass.

Then the transformation requires the following steps for its implementation: (1) an empty class to be added to the program at first; (2) an exclusive component of this class to be added to the oldClass; (3) each method to be moved first to be 'abstracted' by constructing and returning a method which has same signature as a method; (4) each

filed to be moved to be 'abstracted' by defining a field which has same signature as a method; (5) then move the method or the field to the new class.

Finally, the state functions must hold after applying the transformation: (1) a new class called newClass has been added to the program; (2) the class oldClass has a field called 'movement'; (3) all methods or fields defined directly or indirectly in oldClass that are used by a method in now public; (4) the given methods and fields have been mobbed to the newClass and (5) the oldClass delegates invocations of the moved methods or fields to those that exhibit the same behaviour in the newClass.

5.7.1.2 Encapsulation Transformation

The transformation aims to be applied when one class creates instances of another and it is required to weaken the association between the two classes by packaging the object creation statements into dedicated methods. The transformation requires three parameters namely: name of the class to be updated (creator), name of the product class (product) and name of the new constructor method.

For the applicability of the transformation, the condition functions in the source code features as follows: (1) the class *creator* exists and (2) the creator class defines no method and have the same signature as a constructor in the class *product*.

Then, the transformation requires the following steps to be implemented: (1) for every constructor in the product class, a new method called *createProduct* is created in the creator class, (2) all product objects created in the creator class are replace with invocations of the appropriate *createProduct* method expression e with an invocation of the method *createProduct* using the same argument list.

Finally, these state functions hold after applying the transformation: (1) for every product object creation expression in the creator class, a method called *creatorProduct* that creates the same object is added to the creator class and (2) every product object creation is deleted.

5.7.1.3 Wrapper Transformation

The transformation aims to 'wrap' an existing receiver class with another class, in such a way that all requests to an object of the wrapper class are passed to the receiver object it wraps and similarly any results of such requests are passed back by the wrapper. It requires two parameters namely: (1) the name of a single receiver class or a set of receiver classes to be wrapped (client), (2) the name of an interface that reflects how the receivers are used in the client class (interfaceName) and the name of the wrapper class (wrapperName).

For the applicability of the transformation, the condition functions required to evaluate preconditions in the source code features as follows: (1) the given interface must exist and (2) the name for the new wrapper class is not in use.

Then, the transformation requires the following steps to be implemented: (1) the wrapper class is created and added to the program and (2) the wrapper class is used to wrap each of the receiver classes and, consequently, any clients that use these receiver classes are updated to wrap each construction of a receiver class with an instance of the wrapper class.

Finally, the following state function must hold after applying the transformation: (1) the wrapper class has been added to the program, (2) all object references to receiver classes is *client* have been changed to *wrapperName* and (3) all creations of receiver of

objects in the client have been updated.

5.7.2 Transformation Extension on Multimedia Application

The most distinguished feature of multimedia software is the temporal and spatial relationship between multimedia objects. Therefore, the proposed transformations in this area focus on the spatiotemporal relationships. The transformations looked into are the ones to keep the semantics of the multimedia application after changing the syntax. Semantically, the layout and the temporal arrangement of multimedia objects represent the external behaviour of such an application. Any operation, which alters the external behaviour, can result in the alteration of semantics. For the analysis of multimedia application, the essential of the temporal and spatial relations are the most concerns. Therefore, the transformations to be developed for multimedia application will devote to the comprehension of the two characteristics.

5.7.2.1 Abstraction Transformations

Abstraction transformations used for a multimedia application aim to strip off the statements which are irrelevant to the features for which the program is abstraction. From the definition of a transformation, the abstraction transformations should be regarded as partial transformation since the semantics of a program are not preserved completely and only the interested semantics are preserved.

Transformation 5-1 (Spatial_Abstraction Transformation) The transformation is to remove the statements which are irrelevant to spatial properties.

For example, the temporal properties can be removed by this transformation.

Transformation 5-2 (Temporal_Abstraction Transformation) The transformation is to

remove the statements which are irrelevant to temporal properties.

For example, the spatial properties can be removed by this transformation.

5.7.2.2 Replacement Transformations

The replacement transformations are to replace one construct by another construct under the consideration of semantic preservation. This kind of transformations can be used to increase or decrease the abstraction levels of a multimedia application.

For a temporal relation $\boldsymbol{R}$ defined in Table 5-1 on a media object set $\boldsymbol{A}$, $x, y \in \boldsymbol{A}$, there is the following definition.

Definition 5-12 (Inverse Relation) $\boldsymbol{R}^{-1}$ is defined as $\boldsymbol{R}^{-1} = \{(y, x) \mid (x, y) \in \boldsymbol{R}\}$.

For example, in the given relations, the inverse of $\underline{s}$ is $\underline{si}$, i.e., $\underline{s}^{-1} = \underline{si}$.

Based on the definition, the transformation is given below.

Transformation 5-3 (Inverse Transformation) For $x, y \in \boldsymbol{A}$, $x\boldsymbol{R}\,y$ can be transformed as $y\boldsymbol{R}^{-1}x$.

For example, $x\ \underline{s}\ y$ can be changed as $y\ \underline{si}$ x. By this transformation, the kinds of relation operators can be replaced, thereby improving the understandability of program.

Some temporal and spatial relations are implicit because the temporal characteristics are represented as the properties of multimedia objects. To understand the temporal and spatial behaviour more specifically, the transformations are needed to extract the explicit relations from the properties of media objects. The semantic definition of the two temporal relations can be used to deduce such transformations.

Transformation 5-4 (Replace_by_Concrete Transformation) The transformation is used to replace the temporal properties by the concrete temporal relations. If the properties of two multimedia objects satisfy one of the following conditions, then the two objects can be transformed to having the corresponding relations.

- If ((x.start + x.duration < y.start) ∧ Seq(x;y)), then x;y can be transformed as x $\underline{\leq}$ y;
- If ((x.start + x.duration = y.start) ∧ Seq(x;y)), then x;y can be transformed as x **$\underline{m}$** y;
- If ((x.start = y.start) ∧ (x.duration = y. duration) ∧ Par(x;y)), then x||y can be transformed as x $\underline{=}$ y;
- If ((x. start + x. duration > y.start) ∧ Par(x;y)), then x||y can be transformed as x **$\underline{o}$** y;
- If ((x.start < y.start) ∧ (x. start + x. duration < y.start) ∧ Par(x;y)), then x||y can be transformed as x **$\underline{d}$** y;
- If (x.click ∧ (x.start = y.start) ∧ Par(x;y)), then x||y can be transformed as x **$\underline{s}$** y;
- If (x.click ∧ (x.start = y.start + y.duration) ∧ Par(x;y)), then x||y can be transformed as x **$\underline{f}$** y;

5.7.2.3 Absorb Transformation

In Table 5-1, fourteen temporal relations between media are presented including the forward relations and the inverse relations. The properties of the temporal relations for the transformations are gathered as follows.

Definition 5-13 (Transitive Relation) For a temporal relation ***R*** on a media object set ***A***, $x, y \in A$, if $x\boldsymbol{R}\ y$ *and* $y\boldsymbol{R}\ z$ implies $x\boldsymbol{R}\ z$ for all $x, y, z \in \boldsymbol{A}$, ***R*** is a transitive relation.

Based upon the definition, the temporal relations that are relevant to the sequence relation and the parallel relation are transitive relations. Therefore, we can have the absorb transformation according to this property.

Transformation 5-5 (Transitive_Absorb Transformation) If ***R*** is transitive, $\exists x, y, z$ satisfies $x\boldsymbol{R}\ y$ *and* $y\boldsymbol{R}\ z$, when only media x and z are concerned, the media y can be absorbed by replaced as $x\boldsymbol{R}\ z$.

Transformation 5-6 (Elimination_Absorb Transformation) For $x \in \boldsymbol{A}$, $x\mathbf{Seq}x$ can be transformed as x and $x\mathbf{Par}x$ can be transformed as x.

The above two transformations can be used for simplifying program and improving its understandability.

5.8 Program Transformation Catalogue

In this section, the existing program transformations are explored and classified into eight categories. In the current transformation bank, there are about one hundred transformations. A reasonable and clear catalogue is important for the transformation management and operation. Further, the classification is also necessary for the transformation predication and evaluation of their impacts. In order to make the classification clear and reasonable, the criteria are defined as follows.

Hierarchy criterion: according to the language levels of WSL, the transformations are divided as two groups: basic transformations and secondary transformations, corresponding to the WSL levels respectively.

Operation criterion: according to the effects of their operations, the transformations can be classified as eight groups: Insert, Delete, Simplify, Join, Rewrite, Move,

Abstraction and Refinement.

- *Insert*: The transformations in this group insert the program nodes or statement into the program.

- *Delete*: The transformations in this group remove the program nodes.

- *Simplify*: The transformations are used to simplify a program by change its construction rather than simply delete its nodes.

- *Join*: The purposes of the transformations are to merge several constructs into a single construct.

- *Rewrite*: The transformations rewrite the program block according to the rules without removing or inserting the program constructs.

- *Move*: The transformations are used to move the position of program nodes.

- *Abstraction*: The transformations in this group raise the abstraction level of statements.

- *Refinement*: The transformations in this group refine an abstraction specification to a statement.

Strictly, the transformations in Abstraction and Refinement group are the semi-semantics preserving transformations in that the semantics can be lost or added partially during the abstraction and the refinement process.

The catalogue based on the operation of transformations has been adopted in the existing work. The program transformations in the transformation bank are marked as

different groups. This catalogue is useful for the transformation predication or selection since the effects of the transformations can be viewed by a straightforward way.

5.9 Meta-Model of Transformations

The process of devising and composing transformations that introduce source code altering operation in an existing system poses an investigating challenge for the reengineering of such systems. The process is both a top-down for advanced transformations and a bottom-up for the lower level design motifs.

Figure 5-14 shows the meta-model of the transformations, which presents the conceptual relations between transformations.

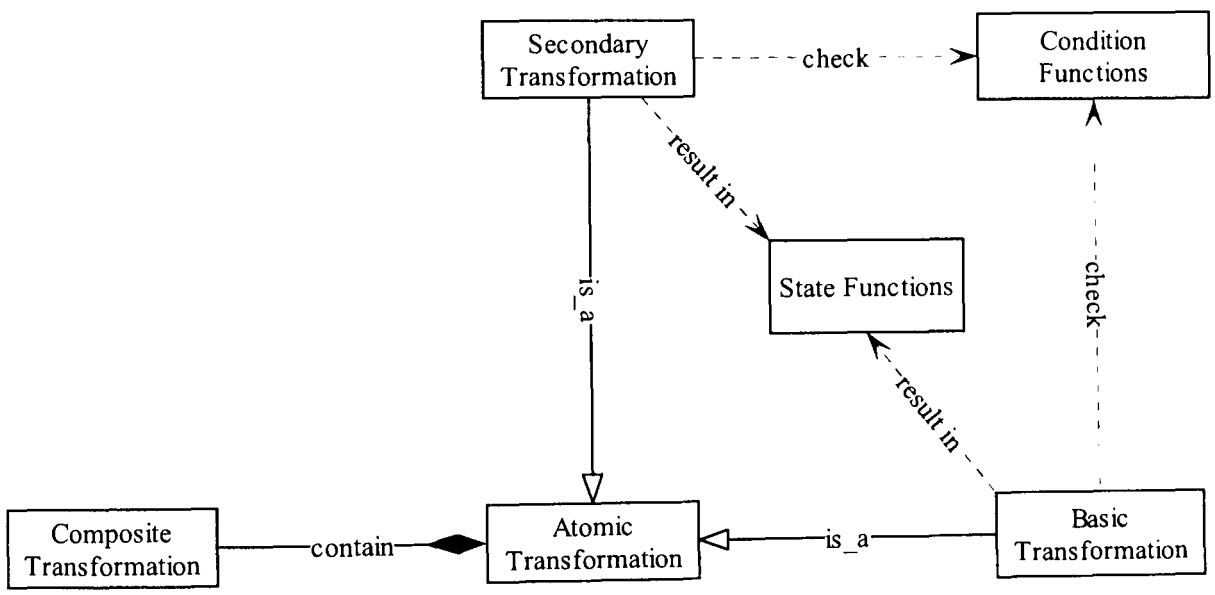

Figure 5-14 Meta-Model of the Transformations

The basic transformations and the secondary transformations are a number of small simple atomic transformation rules called axioms to parts of a program's source code. These axioms are formally proven correct in that they are semantic equivalence preserving transformations. It is presumed that if each axiom preserves semantic equivalence when a whole sequence of axioms ought to preserve semantic equivalence. The secondary transformation is not the composite of the axioms but used for

processing the WSL constructs at the secondary level.

In defining a transformation, the condition functions and the state functions are used to specify sets of preconditions and consequently assertions which should be made about the program. Transformation process is presented with a concise and step-by-step description on how to carry out and implement transformation. Consequently, the atomic transformation can be combined to produce composite transformations. The concepts involved in the meta-model will be specified further in the following sections.

5.10 Mathematical Notations of Program Transformation

In the proposed research, a transformation is stated as a state transformation process which changes software system but preserve the external behaviour. The initial state and final state to applying a transformation, i.e., precondition and postcondition of the transformation are specified by First-Order Predicate Formulas (FOPF) [54]. In addition to the standard logic symbols $\{\neg,\wedge,\vee,\rightarrow,\equiv,=,\exists,\forall\}$, FOPF includes a set of extralogical symbols that specify the various functions and predicates. In the proposed transformation meta-model, the condition functions are used to test whether the initial state is eligible to perform a transformation. In addition, the state functions are used to specify the final state after applying a transformation. Both condition functions and state functions are specified as FOPF. In the next two sections, the two kinds of functions will be illustrated in detail. The mathematical notations, which are necessary to be precise about the effect of a transformation on a program, are used to specify the transformation model.

- P : the program to be transformed .

- T : the transformation based on WSL

- I_P: denotes an interpretation of first-order predicate logic where the universe of discourse comprises the program elements of P and the functions and predicates of the calculus reflect the condition functions as applied to the program P.

- *Pre*(T) : denotes the evaluation of the precondition of the transformation T on the program interpretation I_P. The precondition is written as condition functions

- *Post*(T): denotes the program interpretation I_P, rewritten with the postcondition of the transformation T. The postcondition is written as state functions.

5.10.1 Condition Function and State Function

The condition functions serve the investigative role for performing transformations. They are used as predicates examining whether a specific transformation can be applied in a specific source code context. The condition functions are implemented as the test functions in the FermaT transformation engine. To define the condition functions, it is needed to define the test functions and specify the condition functions as the first order predicate logic formula. The test functions are classified as two types:

Item type testing checks the type of the current node. To serve this testing, two test functions are supplied as follows.

- *@Specific_Type?*(*@Item*, *type*) returns true if the specific type of the current item is the type otherwise returns false.

- *@General_Type?*(*@Item*, *type*) returns true if the general type of the current item is the type otherwise returns false.

- *@Has_Type?*(*@Item*, *type*) returns true if the current item and its children contains the type otherwise returns false.

Program pattern *testing* checks if a block of program matches a pattern on which a transformation can be applied.

- *@IFMATCH?*(*@Item*, *pattern*) returns true if the current item matches the pattern variables otherwise returns false.

- *@Target_Match?*(*target*, *pattern*) returns true if the target and the program variable match an expertise rule which is stored in the transformation engine as a knowledge otherwise returns false.

Pattern variables in the schema are either matched against the current value of the corresponding variable or, if the current value is < > then the corresponding variable is set to the matched item or list of items.

Within the pattern checking function, the pattern variables are allowed:

- '~?*x*' matches any item and puts the matched result into variable x;

- '~**x*' matches a sequence of zero or more items and puts the result into x;

- '~=*x*' matches the current item against the value of the expression e.

The Condition Functions are expressed as the combination of the testing functions by logic operators.

Within *Meta*WSL the condition @Trans?(name) tests if the given transformation is valid at the current position and the statement @Trans(name, data) will apply the given

transformation at the current position, passing data as the additional argument. For example, 'data' might be the new name to use for a procedure renaming transformation).

State functions are used to express the final state of a transformation by FOPF. The State Functions of a transformation denotes an interpretation of first-order predicate logic where the universe of discourse comprises the program elements of *Post*(*T*) and the functions and predicates of the calculus reflect the transformation as applied to the program *P*.

5.11 Construction of Transformation Bank

The transformation bank is the container to store and manage the transformations in the transformation engine. The metadata of the bank defines the properties of the transformations. Each transformation has the properties for the further implementation and predication. Internally, the transformation bank adopts the following attributes as registration information of each transformation for the management and the prediction approach.

- *Index Number* (*ID*): the ID of a transformation
- *Name*: the external name of transformation
- *Proc_Name*: the identifier of transformation procedure
- *Keywords*: the keywords to specifying the usage of a transformation, such as absorb, simplify and so on
- *Applied Specific Type*: the specific type on which a transformation can be applied
- *Category*: the category which the transformation belongs to
- *Help*: the help information to describing the usage of a transformation

- *Prompt*: the prompt when applying a transformation
- *Condition Function*: the precondition of a transformation which consists of condition functions by the first order predicate logic
- *State Function*: the postcondition of a transformation which consists of condition functions by the first order predicate logic
- *Impacted qualities*: the qualities which can be effected by the transformation
- *Algorithmic description*: the text-based description of the transformation algorithmic

5.12 Transformation Composition

The transformations can be implemented with different effects, which consequently can result in the change of source code features. However, a single transformation is not enough to achieve some targets, such as 'high maintainability'. One or more transformations need to collaborate to realise the transformation goal. When building such collaboration of transformations, it is also crucial to determine which transformations are mutually dependent and which transformations have to be applied sequentially. The composition can be implemented into a process according to the conditions of transformations, such as the functions defined in the above sections. The composition patterns can be defined by the algebra operators as follows.

- **Sequential Composition (;)** is where a sequence of transformations is applied one after the other. T_1; T_2 denotes the sequential composition of two transformations T_1 and T_2, If the application of T_1 terminates then the execution of T_2 follows that of T_1. The sequential dependence with this kind of relationship can be detected according to the condition functions and state functions of the transformations, i.e., *Pre* (T_2) = *Post* (T_1).

For the parallel composition, there is a constraint to determine the eligibility of the relation, i.e., the constructs of a program. Definition 5-14 gives the definition of the 'independent relation' related to the constraint.

Definition 5-14 (Independent Relation) If two transformations can be applied to the different positions which are at the same level on AST of a program and the execution order of the transformations does not affect the result, then the transformations are independent against each other.

- **Parallel Composition** (||) is where a set of transformations are performed in parallel. $T_1||T_2$ expresses that T_1 and T_2 can apply in parallel on different program blocks, which are independent to each other.

If two transformations are independent, then the two transformations can be executed 'in parallel'. The parallel execution is different from the concurrent execution in the definition. To execute two transformations in parallel is not to execute them at the same time but means the two transformations are not executed depending on different parts of the program.

5.13 Summary

This chapter explored the language extension of WSL as discussed about the needs. The extensions on object-oriented program and multimedia program are performed by utilising the existing construct levels of WSL and semantic theories. In addition, the chapter addresses the transformation bank extension by developing new transformations, providing a transformation catalog and presenting the method of transformation management. The extension will support the proposed transformation prediction. To summarise, the following contents are involved in the chapter.

- In order to support the target driven approach, new functions are added in MetaWSL, such as @Get_Trans(POSN,SC) to get all of the available transformations for a position of an AST.

- The fix point theory is used to extend the denotational semantics of WSL with object-oriented semantics so that the new constructs can be consistent with the current language semantically and syntactically.

- New object-oriented constructs added include class, object, reference, and inheritance.

- By studying the features of multimedia application, WSL is extended to present multimedia presentation. The extension is based on the three relations, i.e., logic relation, temporal relation and spatial relation, which are the fundamental features of the multimedia application.

- The secondary transformations match the needs for transformations based on the extension of WSL.

- The precise transformation catalog can be used as knowledge or heuristics for the prediction of transformation.

- The meta-model and the mathematical notation of transformations will be used for constructing the transformation path.

- The definitions of the two composition relations show how the transformations are executed together.

Chapter 6
Algorithm of Program Transformation Step Prediction

Objectives

- To model the transformation step prediction as a search problem
- To explore the relationship between transformation, metrics and target
- To give a metrics based prediction algorithm using a heuristic search algorithm
- To incorporate expertise into the prediction algorithm
- To exploit domain features for implementing the prediction algorithm

6.1 Introduction

In the previous chapters, a number of dedicated software metrics are presented and the concept of reengineering target and representation is given with the correlation of the metrics. WSL is extended with advanced features, such as object-oriented features and its applications are extended into the other domains, such as multimedia domain. The

extension of transformations as well as the management mechanism of the transformation bank is presented. Based on the above work, this chapter discusses an algorithm to predicting source-code transformations for specific reengineering targets. The algorithm takes three scenarios into account, i.e. the one based on software metrics merely, the one incorporating with expertise and the one dealing with specific domain features.

6.2 Regarding Transformation Prediction Problem as a Search Problem

In the transformation bank, a large number of transformations are developed for different purposes. When implementing the transformations, software engineer selects the transformation candidates from the transformation bank according to her/his experience and the transformations' usages. For the one who is new to the transformations, facing to the great amount of transformations, she/he could face the difficulty to determine which transformations should be selected and how the steps of the transformation execution should be determined.

Therefore, the capability of a transformation engine to assist and present clues to the users will be useful to improve the transformation implementation's efficiency. Such a functionality of the transformation engine is called as *prediction*. To provide the useful information guiding the transformation process, the transformation engine is supposed to determine the suitable candidates and predict the impact of the transformations. By the information, the efficiency of the system reengineering by transformation can be improved.

How to predict the required transformations is an important issue of the proposed

approach. The prediction in practice is to search the applicable transformations, determine which transformations should be applied, give the possible execution sequence of the transformations and provide the information to the user. Therefore, the prediction can be performed via a search operation to obtain the answer.

Search techniques usually involve a heavy computational burden. However, heuristic search strategies that use some kind of additional (heuristic) information can reduce these computational costs for many problem instances. In the proposed approach, the prediction process is performed as an instance of a heuristic search algorithm. Before applying the algorithm, the relevant entities and relations in the predication framework need to be elaborated.

6.3 A Model of Target-Metric-Transformation Correlation Representation

The proposed transformation process leads to achieving a set of reengineering targets which can be measured by software metrics. Figure 6-1 presents an alternative view of the MOTMET, which was introduced in Section 3.3.2, to show the specific relations between the entities in the MOTMET.

In the figure, the relations are modelled as a directed graph (digraph) D represented as a set of 3-tuple elements $<N, E, L>$ where N is a set of vertices or nodes, divided into target nodes, metric nodes, transformation category nodes and transformation nodes. The set of nodes or vertices is called the vertex-set of D, denoted by $N=V(D)$. The top target is represented as the entry node. E is a set of edges or arcs which connect the ordered pairs of the nodes. The list of arcs is called the arc-list of D, denoted by $E=A(D)$. If n_i and n_j are vertices, then an arc of the form $n_i n_j$ is said to be directed from n_i

to n_j, or to join n_i to n_j. In this case, node n_j is said to be a successor of node n_i and node n_i is said to be a parent of node n_j. There are four kinds of arcs: (1) arcs which connect target and target factors; (2) arcs which connect targets and metrics; (3) arcs which connect transformation categories and targets. This kind of arcs provides the heuristic information for support the search algorithm; (4) arcs which represent ascription of a transformation to its category.

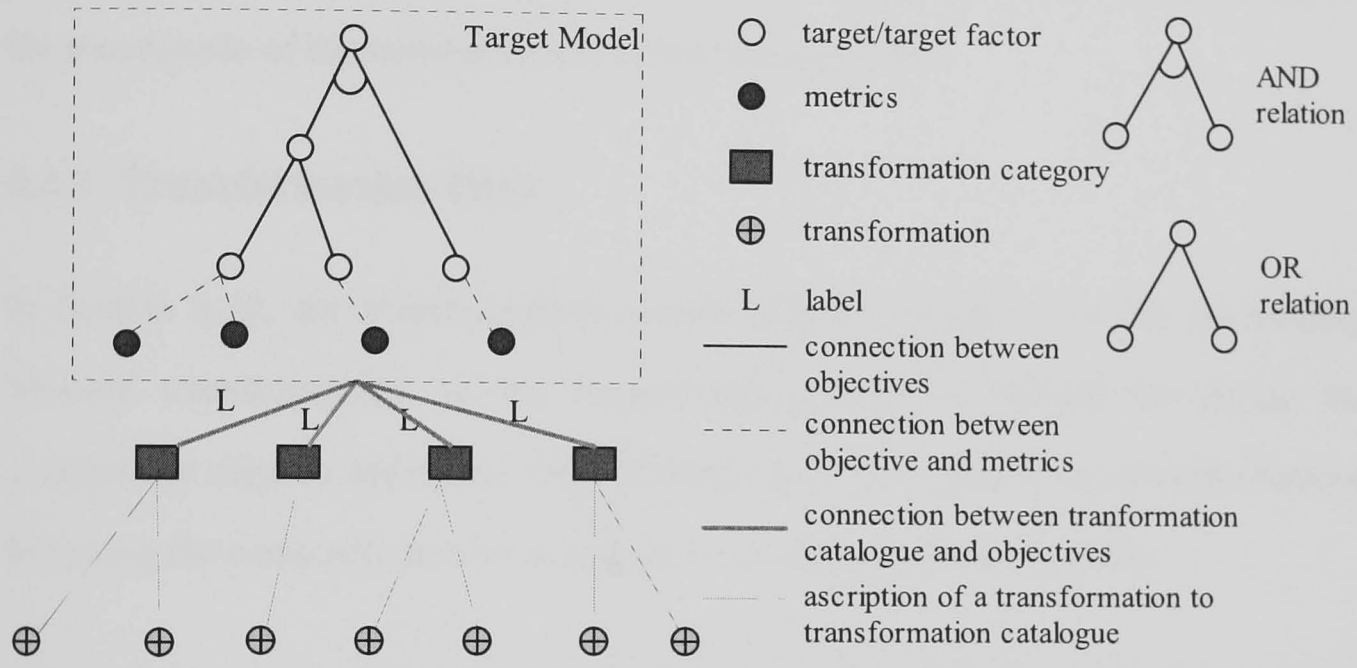

Figure 6-1 Target-Metric-Transformation Correlation Representation

Finally, L is a label of $N \times E$ which assigns to each node of D and to each edge an impact rule as it will be elaborated later. The labels have different meanings corresponding to different kinds of arcs. The meaning and the way to calculate the value of a label will be explained later in this chapter.

6.4 Transformation Step Prediction Algorithms

The transformation step prediction algorithms aim to provide the means for the reengineering target driven transformation process. The result is a set of solutions which include the transformation execution sequences for the desired target. The

resources used in the approach can be not only the program to be studied, but also the expertise for performing reengineering. The transformation prediction algorithms are presented by utilising those resources on different aspects. The basic algorithm is a metrics based prediction algorithm. The second algorithm is to incorporate expertise which is represented as rules. When the prediction algorithm is applied in a specific domain, the domain features are necessary for leading the process.

This section formulises the problem and gives the details of the algorithm, finally gives the pseudocode of the transformation prediction algorithms.

6.4.1 Transformation Path

In Section 5.12, the transformation composition is discussed and the relationship between transformations in the composition process is defined by giving the composition algebra operators. Transformation process is to execute transformations following the composition rules to complete a set of transformation task.

Assuming to be given a specific target, the transformation process should be performed toward the target. The process is implemented as a transformation sequence. It is called the transformation path which is composed of a sequence of transformations connected through the composition algebra. The formal definitions are given as follows.

Definition 6-1 (Transformation Path) Given a program P, Transformation Path Λ on P is t_1 ***op*** t_2 ***op*** ... ***op*** t_n where $t_i = T_i(parameters_i)$, $T_i(i = 1, \cdots, n)$ is a program transformation on P, the $parameters_i$ specify the position where T_i is applied and the arguments T_i needs for its execution, *op* can be the sequential composition operator ';' or the parallel composition operator '||'.

According to the definition of sequential composition, if two transformations are connected by the sequential operator in a transformation path, i.e. $t_1;t_2$, that means after the program P is transformed to P' through t_1, t_2 is used on P' as the succeeding transformation. If two transformations are composed by the parallel operator in a transformation path, i.e., $t_1 \| t_2$, that means t_1 and t_2 are applied respectively on the two independent blocks of the program P at one transformation step.

Definition 6-2 (Transformation Step) A transformation step in a transformation path is the execution of a parallel composition of transformations or the execution of single transformation. Alternatively, a transformation path Λ can be denoted as $\tau_1,\cdots,\tau_m$, where $\tau_i(i=1,\cdots,m)$ is a transformation step on the path.

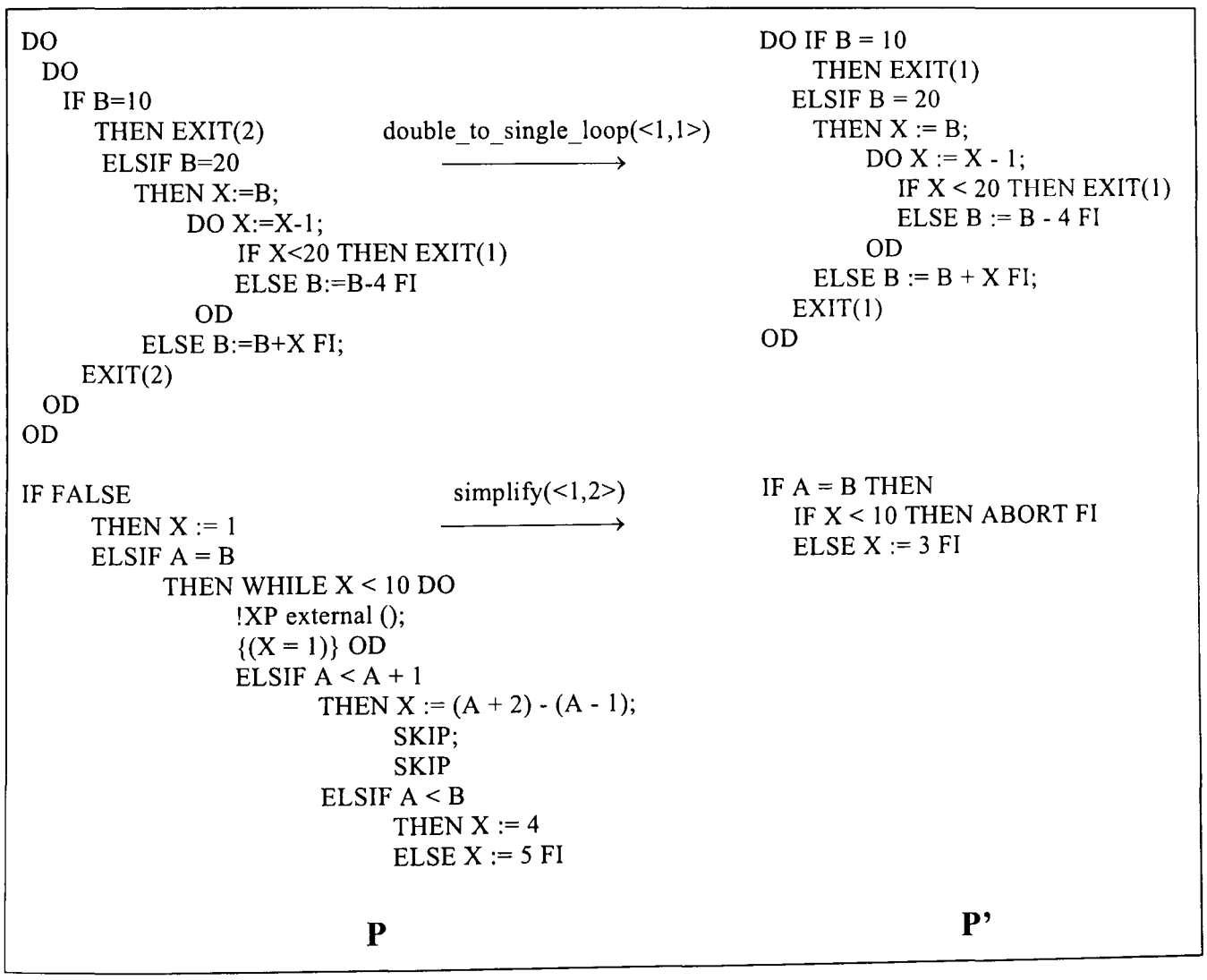

Figure 6-2 Example of Transformation Parallel Composition

The transformation execution composed by the parallel relation is regarded as a single step of execution while the sequential composition of two transformations is regarded as two individual steps. Therefore, the number of the transformations contained in a transformation path may be greater than the number of the steps. For example, shown in Figure 6-2, a transformation step on program *P* includes two transformations composed by the parallel relation, i.e.,

double_to_single_loop(<1,1>) || simplify(<1,2>)

6.4.2 Problem Formulation

Conceptually, the target driven transformation process can be modelled as a transformation path t_1 ***op*** t_2 ***op*** ... ***op*** t_n and a sequence of states, $s_0, s_1, \cdots, s_i, s_{i+1}, \cdots, s_m$, where state s_{i+1} is yielded from state s_i by a transformation step and both of the versions are semantically equivalent. Each state s_i is quantified by a set of metrics chosen from the target model and represents the outcome of the system at the transformation step τ_i.

Based on the constructed target model and software features of the source code versions, the aim is to quantify the impact of the transformation step τ towards the selected target in terms of the target computation, which is referred to a formulation of target score. For a selected target, each of the transformation t_{ij} is associated with a target score, ts_{ij} which relates to the impact of transformation on the program towards the desired target for the given system. The computation of target score will be exploited later.

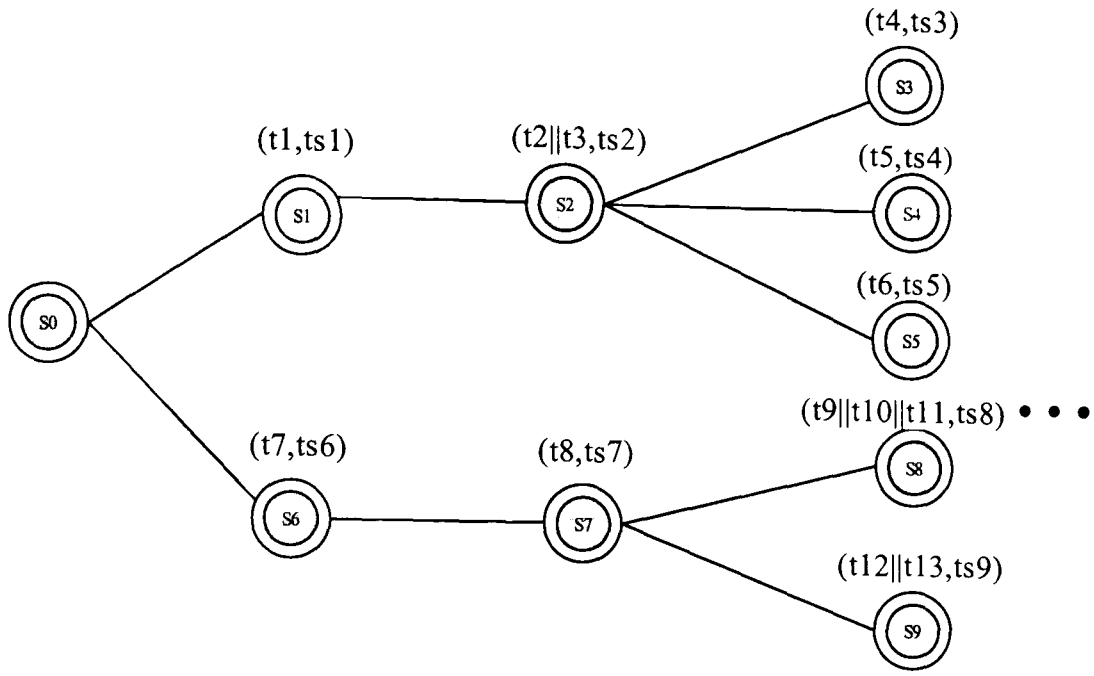

Figure 6-3 Target Driven Transformation Process Model

Definition 6-3 (Transformation Process Model) Transformation Process Model (TPM) is a tree structured model composed of possible transformation paths which have the positive impacts towards the reengineering target.

The process model is shown in Figure 6-3 which can be modelled as a weighted tree. Let $G = (N(G), A(G))$ to be a tree. $N(G)$ to be the set of nodes in G that are state and $A(G)$ be the set of arcs in G that are the transition of states. Thus, $A(G) \subseteq N(G) \times N(G)$. The initial given problem is represented by a unique node in G called the state root s_0. Furthermore, each node has value with a vector $<\tau, ts>$, where τ is a transformation step and $ts \neq 0$ is target score which is to value the impact of the transformation step with respect to the desired target. Any state whose ts is equal to 0 will not be added into the tree. In G, a transformation path is the path from the initial state root to a node in the tree.

In the proposed approach, a set of transformation sequences are predicted that may yield the desired properties for the reengineered system. At the nth step the various

transformations 'in competition', $t_1, t_2, \cdots, t_k$ are applied to the sequence (S_n) of a metric space E. The transformation candidates $t_1^{(n)}, t_2^{(n)}, \cdots, t_k^{(n)}$ are obtained. Then, one of them is predicted by the transformation engine. The one must be the best one at this step. Such a problem can be formulated as a search in the space of alternative transformations based on the transformation process model.

Given Problem:

- A target model representing the target which the program transformation process aims to;
- A problem state-space, represented as a finite tree, where the states of a program are represented the nodes and the transitions over the states are the arcs;
- A root node in the tree to represent the initial state;
- A vector of transformation step and impact-valued criterion associated with each node in the tree;
- A preference among paths on their impacts.

Find Solution:

- A list of ranked transformation paths as solutions (potential target-driven transformation steps) in the tree.

From the formulation, there are three important factors as follows.

(1) How to compute the target score to evaluate the transformation impact towards the desired target.

(2) How to construct TPM for a program. This is a key for the prediction problem.

The model graph is generated according to the examination of the applicable transformations for the current state based on the heuristic functions. The heuristics are based on several rules obtained from experience, expertise and domain features.

(3) How to find the solution for the target. After constructing the transformation process model, the prediction will rely on the constructed paths and the generated value of target score.

The above three questions will be addressed respectively in the next few sections. After giving the solution of them, the prediction algorithms will be given in pseudocode.

6.4.3 Transformation Impact Function

As aforementioned, source code changes occur as a transformation is applied in the proposed framework. Every transformation can alter the system into a new version. The semantics of the different versions over the transformation process are equivalent according to the essential requirement of transformations. On the other hand, the process of transformation can result in a new program with altered source code features. The TM systematically models source code features that are related to a specific reengineering target. Moreover, TMs provide a guideline on how to measure the desired system features by the selected metrics. The target model thus gives a means to software engineer for identifying the optimal combination of transformations that may have the highest likelihood to yield the desired target during the software reengineering process.

In order to differentiate the versions in terms of their impacts on software features,

every transformation in the predict results must conform to the following two rules: Rule 6-1 and Rule 6-2.

Rule 6-1 Every transformation ***t*** on a target driven transformation process model causes at least one change in a selected metrics.

Rule 6-2 The change caused by a transformation is quantified by the identified metrics modelled as leaves in the target model.

Formulated in the previous section, target score is used to evaluate the impact of a transformation step on the desired target. The target score $ts(t_i)$ of transformation t_i can be calculated by a function, called impact function $impact(t_i)$ is defined as $\frac{Benefit}{Cost}$, which have the best impact on the target by maximising *Benefit* and minimising *Cost*.

Any transformation performs the operations such as adding, modifying and/or deleting source code entities. Such operations affect the efficiency of the transformations' implementation. The entities referred here are the nodes on the AST of a transformed program. By this consideration, the *Cost* for applying a transformation t_i, which is used to estimate the transformation cost, can be measured as Formula 6-1. The formula is constructed according to the cost of adding, modifying and deleting nodes. It is normalised in order to make the value within the united range.

Formula 6-1 Cost Function:

$Cost(t_i)$ = Normalisation of (number of nodes added + number of nodes modified + number of nodes deleted)

$$= \frac{\#added\_nodes + \#modified\_nodes + \#deleted\_nodes}{\sqrt{(\#added\_nodes + \#modified\_nodes + \#deleted\_nodes)^2 + 100}} \in [0,1)$$

In order to measure the *Benefit* for a potential transformation to be applied, a summary of proposed metrics needs to be evaluated. There are two kinds of metrics related to a determined target. One is the metrics that their decreasing values provide more benefit and the other one is the metrics that their increasing values provide more benefit. Since the range of different metric values could be quite different, it is necessary to normalise them into the same range. Regarding to the two considerations, the *Benefit* can be defined as follows by extending Formula 4-1 which was made according to the AND/OR relations of the target model and the positive or the negative effect of a metric on a target. Formula 6-2 gives the more specific formula with the detailed normalisation function to compute the benefit of a transformation.

Formula 6-2 Benefit function:

$$Benefit(t) = \begin{cases} \sum_{i=1}^{p} c_i\omega_i - \sum_{i=n+1}^{q} c_i\omega_i & \begin{array}{l} if \quad AND(\tau, \tau_1, \cdots, \tau_p, \tau_{p+1}, \cdots, \tau_q) \\ \wedge m_1, \cdots, m_p \quad is \quad positive \wedge m_{p+1}, \cdots, m_q \quad is \quad negative \end{array} \\ -\min(\omega_1, \cdots, \omega_p) & \begin{array}{l} if \quad OR(\tau, \tau_1, \cdots, \tau_p) \\ \wedge \min(m_1, \cdots, m_p) \quad is \quad negative \end{array} \\ +\min(\omega_1, \cdots, \omega_q) & \begin{array}{l} if \quad OR(\tau, \tau_1, \cdots, \tau_q) \\ \wedge \min(m_1, \cdots, m_q) \quad is \quad positive \end{array} \end{cases}$$

where $\omega_i = \dfrac{m_i' - m_i}{\sqrt{(m_i' - m_i)^2 + m_{i0}}}$, m_i' is the value of the metric after applying the transformation t_i; m_i is the value before applying the transformation; m_{i0} is the initial value of the corresponding metrics on the transformed program before doing any transformations. Therefore, $m_i' - m_i$ are used to evaluate the effect of the transformation t_i.

The impact of a transformation step is calculated by the quotient of *Benefit* and *Cost* according to the formulae. The value of the impact is the target score of the

transformation step. As stated in Rule 6.1, a transformation alters a system state. The more features that are altered positively by a given transformation towards a desired target as this is modelled by the target model, the higher the target score and the likelihood that the transformation can contribute towards the desired targets. If a target score of a transformation step is equal to 0, i.e., no any benefit to apply the transformation step, then the step will not be added into the transformation process model.

For example, given the target model shown in Figure 4-2, the values of the selected metrics before τ and after τ in the model are listed in Table 6-1.

	NCNB	NON	McCabe	CFDF	RNC
Before τ	40	200	10	28	3
After τ	35	186	8	24	2
Difference	-5	-14	-2	-4	-1

Table 6-1 Example of the Altered Metric Values

Assuming that the transformation step τ (1) adds 10 nodes on AST of a program, (2) deletes 15 nodes and (3) modifies 8 nodes. Therefore, the cost value is equal to:

$$\text{Cost} = \frac{10+15+8}{\sqrt{(10+15+8)^2+100}} \approx 0.957,$$

According to the variation of the metrics, the benefit value is equal to:

$$\text{Benefit} = -\left(\frac{-5}{\sqrt{(-5)^2+40)}} + \frac{-14}{\sqrt{(-14)^2+200}} + \frac{-2}{\sqrt{(-2)^2+10}} + \frac{-4}{\sqrt{(-4)^2+28}} + \frac{-1}{\sqrt{(-1)^2+3}}\right) \approx 2.921,$$

The target score of τ, i.e., the impact value of the transformation τ on the specific target is: $ts(\tau) = \frac{Benefit}{Cost} = \frac{2.921}{0.957} \approx 3.055$.

6.4.4 Heuristic Function

Once the target driven transformation process model is constructed, the preference transformation path can be predicted by comparing the target scores. However to construct such a model is another mission to achieve the goal. When any transformation path in the model is to be expanded, the transformation which yields the next state, i.e. the state node in the model, must satisfy the condition that the transformation must be applicable for the program.

To find such a transformation is a search problem whose search space is all of the transformations in the transformation bank. With the request for the desired target, the search is restricted to find the transformations which contribute the fulfilment of the target. It is not ideal to add all of the applicable transformations into the transformation model in that some of them might not positively impact the target even hurt the target. A big scale transformation model involves a heavy computational burden. It is necessary to narrow the search space and prune the branch of the model graph. However, heuristic search strategies that use some kind of additional (heuristic) information can reduce these computational costs for many problem instances.

To define the heuristic function, it is needed to explore the knowledge embedded in the proposed framework. In general, three kinds of information can be considered.

- The usage of transformations
- The expertise using transformations
- The domain features of the application

The first one is obviously useful for narrowing the search space and can be used as a

kind of heuristic. Expertise obtained from the practical work is also an important heuristic because it is quite straight for the certain target. In addition, domain feature is a crucial factor which can not be ignored when dealing with a domain specific application.

For the nature of the three kinds of heuristics, the algorithms of the transformation model construction the prediction process should address the three aspects according to the status of program. Therefore, there are three cases of the algorithm to be proposed, namely the metrics based algorithm, the expertise incorporated algorithm and the one dealing with the specific domain application.

6.4.5 Metrics Based Prediction Algorithm

The metrics based prediction algorithm utilises the basic heuristic, i.e. the usage of transformations, to expand the transformation process model. The problem to address is what transformations and code changes should be applied to improve the corresponding metrics and therefore the corresponding reengineering target. An intuitive solution is to identify which transformation (or a set of transformations) allows changing the value of a particular metric (or a set of metric). To respond to such a question, two steps are needed to be considered: (1) to propose a catalogue of transformations as a predefined set of transformations; and (2) to analyse the impact of each transformation on the predefined set of metrics. The first step was completely discussed in Section 5.8 and a comprehensive list of transformations is provided in Appendix C.

Each transformation in the transformation bank has a particular development purpose and usage. Some transformations are used to remove code redundancy, some can raise the level of abstraction, some enhance the reusability, some have a positive effect on the performance and so forth. The relations can be captured by experience and the

purpose to develop a specific transformation.

These transformations modify the structure of a program which will possibly modify in a positive way the values of the metrics related with the quality being improved. The potential impact of applying each transformation on metrics is shown in Table 6-2 . Note that the increase effect '+' means the transformation increases the metric's value, the decrease effect '–' means the transformation decreases the metric's value, the uncertain effect '+/–' means that the transformation can decrease or increase the metric's value and the non-effect '*N* ' means that there is no impact. The knowledge obtained from the connection can be used as an essential heuristic which contributes the decision of predicting the proceeded transformations.

In terms of the relations between metric and target, i.e., the positive metric-target relation and the negative metric-target relation detected in a target model, another two relations with regard to the transformations impact on a target can be described as Figure 6-4.

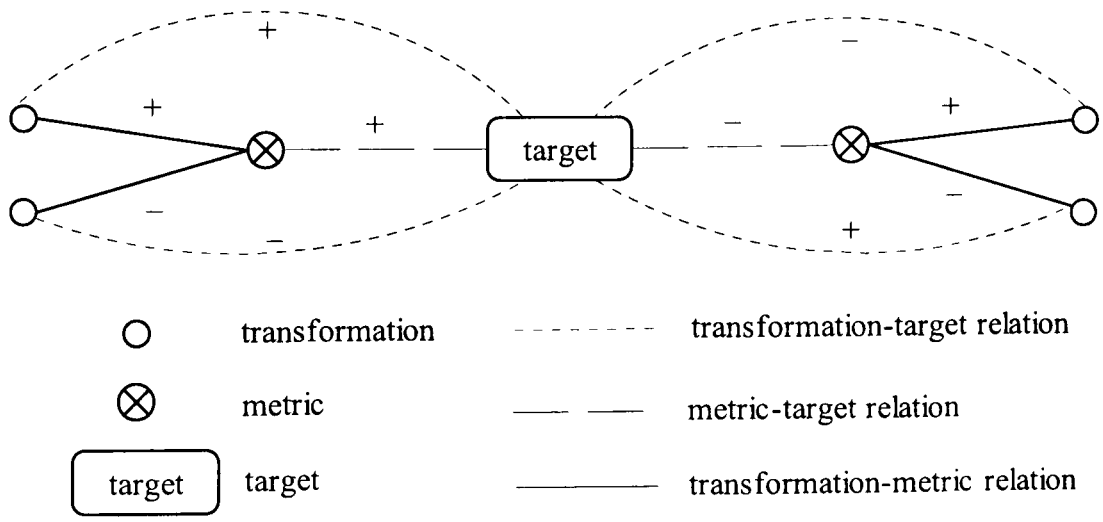

Figure 6-4 Impact Relations between Target, Metric and Transformation

Trans. / Metrics	*Insert*	*Delete*	*Simplify*	*Join*	*Rewrite*	*Move*	*Abstraction*	*Refinement*
NCNB	+	-	-	*N*	+/-	*N*	-	+
McCabe	+/-	+/-	-	*N*	+/-	*N*	*N*	+
WOC	+	-	+/-	*N*	+/-	*N*	-	+
NON	+	-	-	*N*	+/-	*N*	-	+
CFDF	+	-	-	*N*	-	*N*	-	+
RNC	*N*	*N*	-	+	-	*N*	-	+
ABST-LOC	+/-	+/-	+/-	*N*	+/-	*N*	-	+
ABST-STAT	-	+	+	*N*	+	*N*	-	+
ABST-CFDF	-	+	+	*N*	+	*N*	+	-
ABST-VOC	*N*	*N*	*N*	*N*	+	*N*	+	-
NMI	+	-	-	-	-	-	*N*	*N*
DIT	-	+	-	-	-	-	-	+
CBO	+	-	-	-	-	+/-	-	+
WMC	+/-	+/-	+	*N*	+/-	*N*	*N*	+
NVC	+/-	+/-	+/-	-	*N*	-	-	+
WOIL	-	+	+	-	+/-	*N*	+	-
HIL	-	+	+	-	+/-	*N*	+	-
AMS	+/-	+/-	-	-	-	*N*	-	+
SD	-	+	+	*N*	+/-	*N*	-	+
ETL	-	+	+	*N*	+	*N*	-	+
NEI	+	-	-	*N*	-	*N*	*N*	*N*
PSR	+	-	-	*N*	-	*N*	*N*	*N*
PTR	+	-	-	*N*	-	*N*	*N*	*N*
ANFC	+	-	-	+	-	-	*N*	*N*
ANSDF	+	-	+/-	+	-	-	*N*	*N*
ANSFF	+	-	-	-	-	-	*N*	*N*
NCIF	+/-	+/-	-	-	-	-	*N*	*N*
OSC	+	-	-	-	+/-	-	-	+

Table 6-2 Impact of the Transformations on Metrics Suite

Transformation's positive impact on a target factor: a transformation t can positively impact a target factor measured by metric m if t deceases the value of m and m has a negative relation with target Γ, or t increases the value of m and m has a positive relation with target Γ.

Transformation's negative impact on a target factor: a transformation t can negatively impact a target factor measured by metric m if t deceases the value of m and m has a positive relation with target Γ, or t increases the value of m and m has a negative relation with target Γ.

Transformation's uncertain impact on a target factor: a transformation t has the uncertain impact on a target factor measured by metric m if t could decrease or increase the value of metric m.

Transformation's non-impact on a target factor: a transformation t has the non-impact on a target factor measured by metric m if t could decrease or increase the value of metric m.

Given a target model and the table presenting the relations between transformation and metrics, the Transformation Qualification Score (TQS) is used to weight the relations.

Formula 6-3 (Transformation Qualification Score) Given a target Γ correlated with a set of target factors which are measured by metric set M and a transformation category $T, TQS(T) = 1 - \frac{\#negative\_impact}{n}$, where n is the number of metrics in M, $\#negative\_impact$ is the number of the target factors which are negatively impacted by T.

Rule 6-3 $\forall t \in T, TQS(t) = TQS(T)$, where t is a transformation, T is the transformation category which t belongs to.

Rule 6-4 A transformation can be added into a transformation path if and only if $TQS(t) \geq 0.5$.

The Rule 6-4 gives the guidance to narrow the search space and construct the transformation path. Therefore, the transformation path should contain the transformations which positively impact the metrics towards a target.

The metrics based prediction algorithm is carried out only based on the above rules to construct the transformation model without taking the expertise into account. By modelling it as a searching algorithm, the search space of the problem could be all of the transformations stored in the bank. The algorithm can be terminated according to a predetermined number of steps. When the transformation steps have reached the number, the transformation model expansion should terminate.

Another situation to stop the model expansion is the case when there is no program state node added into the model yielded by a transformation step. This is caused because the succeeding transformation step cannot alter the values of any metrics in the target model, i.e., its benefit or target score is equal to 0.

By using the basic heuristics and this algorithm, the desired result of the transformation step prediction is the transformation path which has the highest summation of the target scores. It can be formulated as Rule 6-5.

Rule 6-5 (Best Predicted Transformation Steps) Give a transformation model containing several transformation paths $\Gamma_1, \Gamma_2, \cdots, \Gamma_i, \Gamma_{i+1}, \cdots, \Gamma_n$ which have m steps with the vector $< t_{ij}, ts_{ij} > (i = 1 \cdots n, j = 1 \cdots m)$ as their labels, the best predicted transformation steps $\boldsymbol{P}$ is the sequence of the transformations on the path Γ_q, where

$$\sum_{j=1}^{m} ts_{qj} = \max_{h=1}^{n}(\sum_{j=1}^{m} ts_{hj}).$$

For example, the predicted results included in Figure 6-5 are ranked as follows.

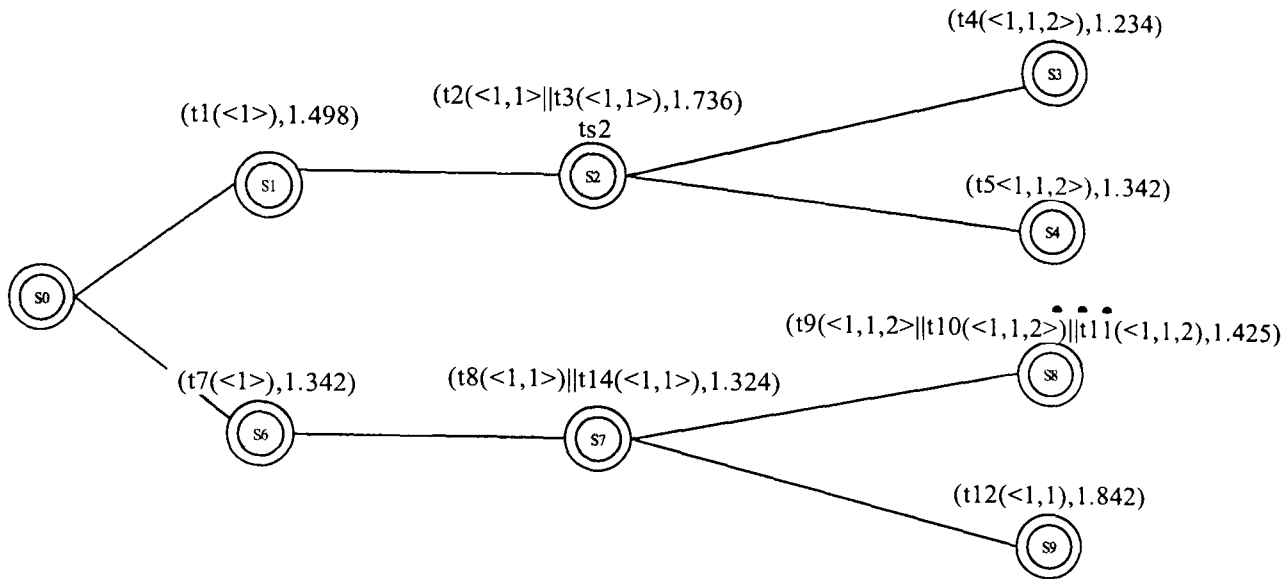

Figure 6-5 Example of Transformation Process Model

TP2 = t1(<1>) ; t2(<1,1>) || t3(<1,1>) ; t5(<1,1,2>). TS2 = 4.576;

TP4 = t7(<1>) ; t8(<1,1>) || t14(<1,1>) ; t12(<1,1>). TS4 = 4.508;

TP1 = t1(<1>) ; t2(<1,1>) || t3(<1,1>) ; t4(<1,1,2>). TS1 = 4.468;

TP3 = t7(<1>) ; t8(<1,1>) || t14(<1,1>) ; t9(<1,1,2>) || t10(<1,1,2>) || t11(<1,1,2>).

TS3 = 4.091;

From the generated transformation paths, the best solution is T2 whose target score is 4.576.

6.4.6 Incorporating Expertise into Prediction Algorithm

In software design, the term pattern has been imported from architecture to describe an application of an expert solution to a common problem in context. Learning the pattern includes understanding the context, the problem, the solution and its merits and

demerits relative to other solutions. Patterns have been adopted enthusiastically by software practitioners because a pattern is an effectively transferable unit of expertise. The vocabulary provided by patterns is also an aid to discussion and clear thought, by experts as well as novices. Importantly, patterns are small and specific enough for the community to validate them effectively. The same benefits will accrue – and possible be even more important – from the identification of program transformation patterns, gained from the expertise.

Different from design pattern, the program transformation patterns obtained from the expertise refer the rules followed by the software engineer when they perform the transformation process. These rules may include the methods of how to select transformations and how to execute transformations. Incorporating these rules as heuristics to establish transformation path and driven the automation of transformation prediction, the searching space of the transformations can be narrowed properly. A list of rules will be given as follows.

Expertise Rule 6-1 (Action System) The program transformation selected for a given action system should implement the heuristics for restructuring action systems. The restructuring steps can include the following steps.

(1) Delete unreachable code;

(2) Remove the tail recursion in an action which calls: by introducing a double-nested **DO**...**OD** loop and replacing the self-calls by **exits**. Further transformation are then attempted to reduce the double loop to a single loop;

(3) Simplify all **IF** Statements which contain calls;

(4) Simplify action bodies to merge calls and remove recursion;

(5) Eliminate actions which are only called once;

(6) Shrink the action by creating procedures from blocks of code;

(7) Remove the last action;

The set of operations has been integrated in a united program transformation, Collapse_Action_System. Therefore, if the object of the program is an action system, i.e., the state function is *@ST(@I)* = T_A_S, the candidate of applicable transformation can be Collapse_Action_System only. This expertise rule is suitable for the program containing action system for the targets which need to collapse action system.

Expertise Rule 6-2 (Merging Similar Statements) The similar statements can be merged and converted to a nested statement in the following scenarios.

(1) Two non-nested similar **IF** statements can be merged and converted to a nested **IF** statement and taking the common code out of the two braches of the outer **IF**;

(2) If the statements appear at the end of a loop and also just before the loop, then loop inversion can be applied to merge the two copies of the statement.

As for (1), for example, two copies "c := 1" can be combined by merging the two IF statements as follows.

```
IF a = 1 THEN
   c := 1; exit(2) FI;
IF b = 2 THEN
   c := 1; exit(1);
ELSE d := 3 FI;
```

Transformed

```
IF a = 1 ∨ b = 2 THEN
    c := 1;
    IF a = 1 THEN exit(2) else exit(1) FI
ELSE d := 3 FI;
```

Below the example is for (2)

```
a := 1;
DO
   b := 1;
   IF c = 1 THEN
      a := 1;
   ELSE d := 1;
   FI;
OD;
```

Transformed

```
DO
    a := 1;
    DO b :=1;
    IF c = 1 THEN a :=1;
    ELSE d := 1; FI; OD;
OD
```

In the implementation, the 'absorb' transformations and the 'merge' transformations can be regarded as the candidates.

Expertise Rule 6-3 (Abstracting a Specification) In order to get to an abstract specification cross the abstraction levels, the heuristics can imply the abstracting as the following steps,

(1) Change the data representation to a more abstract representation;

(2) Restructure data by split function;

(3) Remove ghost variables which have no effect on the execution of the program;

(4) Replace references to the concrete variables by references to the abstraction variables;

(5) Construct abstract procedures to replace the blocks of statement.

The abstracting pattern is general for the most cases. It could vary from case to case. In the transformation bank, the transformation Raise_Abstraction is an available for abstracting purpose. In addition, the transformations, such as Abort_Processing, Compute_WP, Delete_Comments, Unfold_Proc_Calls and so forth also can be used as the candidates.

The rules can be used when constructing the transformation model. As the transformation predictor detects both the target and the state of the program match the pattern in the rules, the transformation path can be predicted as the steps proposed in the expertise rules.

6.4.7 Exploiting Domain Features in Prediction Algorithm

It is necessary to take the domain features into account when applying transformations in a specific domain. In the thesis, the domain features behave as the data and the

relationships between data over a domain. In the previous chapters, the multimedia domain is chosen as the studied cases due to its particular data structures. It is impossible to neglect the domain features and treat those applications as the normal procedural or object-oriented program although they can be converted to the extended WSL based programs.

When using expertise for transformation prediction, the knowledge based rules will provide the hint to apply transformations. The instruction is to guide the users which transformations they should choose. Normally, the domain specific transformations should have the high priority when processing its own domain applications. In addition to this case, by considering domain features, the users might be instructed which transformations they should not choose or be recommended for the domain particularly because of the difference between the normal program and the domain specific program.

Therefore, the clue stated above will be considered as another kind of heuristic used for predicting the transformations for the domain specific program. In Section 7.6, a multimedia case is studied to explore the usage of this kind of heuristic.

6.4.8 Pseudocode of Algorithms

In the implementation of the target driven transformation prediction algorithm, the key procedure is to construct transformation model and generate the transformation path list ranked by the summation of the target scores on the paths. The section gives the pseudocode of the algorithm implementation.

Algorithm 6-1 Predict Program Transformation Steps (PPTS)

Description: To generate the ranked transformation paths from the tree-structured program transformation process model.

PPTS (TM, SC, TL) =

Input:

TM: a target model with the desired reengineering target and the included metrics;

SC: a piece of WSL source code;

TL: the determined number of the tree levels;

Output:

TPs: a set of ranked transformation paths;

Variables:

G: a temporary tree to represent a transformation model tree;

i, j: iteration counter;

tp_sum: a string to present transformation step;

ts_sum: a float to present target score sum of transformation path;

node: a node on the tree

paths: a set of vectors including tp_sum and ts_sum of each path;

Method:

```
1   Initialise the tree G, tp_sum, ts_sum, visited_nodes, paths;
2   G := CTPM (TM, SC, TL); // return the created transformation process model
3   j := 0;
4   for (i:=0; i < G.length; i++) {
5           if(G[i].children == NIL) {
6               node := G[i];
7               while (node.parent<>NIL) {
8                   tp_sum := node.trans_step + ";" + tp_sum;
```

```
9                   ts_sum := node.target_score + ts_sum;
10                  node := node.parent;
11              } //end of while
12              paths[j] := <tp_sum, ts_sum>;
13              j := j + 1;
14          } //end of if
15  } //end of for
16  Sorting(paths, ts_sum); // descent sorting the elements in paths by ts_sum;
17  TPs := paths;
18  return TPs;
19 END.
```

The transformation process model *G* is represented as tree structure. It is defined as the following structures.

```
Structure Node{
        string .trans_step;
        float   target_score;
        int parent;
        array(int) children;
};
Tree G := Array of Node;
```

The following algorithms give the method to generate the program transformation model based on the above structure. In addition, there is an extra structure to be defined and used for the value of tree node in the algorithm.

```
Structure QElem{
        string .trans_step;
        float   target_score;
        int num;
};
```

Algorithm 6-2 Create Transformation Process Model (CTPM)

Description: To create the tree that represents the transformation process model

CTPM (TM, SC, TL, PA, AFlag) =

Input:

TM: a target model with the desired reengineering target and the included metrics;

SC: a piece of WSL source code;

TL: the determined number of the tree levels;

AFlag: a flag of prediction algorithm, 'MBP' — Metrics Based Prediction Algorithm, 'TPBP' —Transformation Pattern (Expertise) Based Prediction Algorithm;

Output:

G: a tree presenting the transformation process model;

Constant:

ExpertiseDB: a table to store the expertise rules including target, patterns and transformation steps;

Variables:

q: a temporary link queue to store the tree node;

p, qq: a QElem type element;

i, j, l, k: Iteration;

c: an array to store the child nodes;

T: a temporary tree;

algorithm_flag: a string to identify the algorithm to be used;

posn: the position on AST of SC;

trans: a set of transformations;

target: a target specified by TM;

pattern: a WSL pattern;

Method:

```
1   Initialise q, pp, qq, c, T, trans, target, pattern;
2   T[0].trans_step := " "; T[0].target_score :=0; T[0].parent = -1;
3   qq.trans_step:=T[0].trans_step; qq.target_score:=T[0]. target_score; qq.num := 0;
4   Enqueue(q, qq);
5   posn := <1>; // the first node on the AST
6   i := 0;
7   while (Depth(T) <= TL) {
8       Dequeue(q, qq);
9       if (algorithm_flag == "MBP") {
10          target := Get_Target(TM);
11          pattern := Make_Pattern(posn, SC);
12          if (@Target_Match?(target, pattern) {
13                  trans:= Get_Tran_Steps(ExpertiseDB, pattern, target);
14          } // retrieve matched transformation steps incorporated with expertise
15      else    trans := GetTrans (posn, SC);
16  }
17  else    trans := GetTrans (posn, SC);
18  k := 1;
19  if (trans <> NULL) {
20      for (j := 0; j < length(trans); j++) {
21          ts := Generate_TargetScore(TM, SC); // in Section 6.4.5
22          if (ts == 0) break;
23          T[i].trans_step := trans[j];
24          T[i].target_score := ts;
25          T[i].parent = qq.num;
```

```
26              T[qq.num].children[k] := i;
27              k := k+1;
28              p.trans_step := trans[j];
29              p.target_score := T[i].target_score;
30              p.num := i;
31              p.parent := T[i].parent;
32              p.children := T[i].children;
33              Enqueue(q, p);
34              i := i + 1;
35          }
36      }// end of if
37      if (?@Down)    posn := @Down(posn);
38  } // end of while
39  G := T;
40  Return G;
41  END.
```

Algorithm 6-3 Get Trans (GT)

Description: Get a set of applicable and qualified transformations for the position on the AST of WSL source code.

GetTrans (POSN, SC) =

Input:

POSN: a position on AST of SC;

SC: a piece of WSL source code;

Output:

Translist: a set of transformations;

Variables:

i, j, k, m_1, m_2, …, m_row : Iteration;

col: the number of columns of the temporary matrix;

row: the number of rows of the temporary matrix;

alltranslist: a complete set of transformations;

transforposn: a matrix containing the transformations for the parallel composition

Method:

```
1   @Goto(POSN);
2   alltranslist := Get_all_trans( );
3   Initialise Translist, transforposn;
4   k := 0;
5   repeat {
6       for (i := 0; i <= length(alltranlist); i++) {
7           j := 0;
8         if (@Trans?(alltranslist[i], POSN)) {
9             if (QTT(TM, alltranslist[i], 0.5) {
10                transforposn [k][j] := <alltranslist[i], POSN>;
11                j := j + 1;
12            }
13        } // end of if
14      } // end of for
15    if (@Left?(POSN)   {
16          POSN := @Left(POSN);
17          k := k + 1;
18    } else break;
19  } // end of repeat;
```

```
20  i := 0;   j := 0;
21  //
22 col := Column_Number(transforposn);
23 row := Row_Number(transforposn);
24 for (m_1 := 0; m_1 < col; m_1++)
25       for (m_2 := 0; m_2 <col; m_2++)
26            …
27                 for (m_row := 0; m_row < row; m_ row ++)
28                      Translist[m_row] := transforposn[1][m_1] + "||"
29                                          + transforposn[1][m_2] + "||"
30                                               …
31                                          + transforposn[1][m_row] + "||";
32 return Translist;
33 END.
```

Algorithm 6-4 Qualify Transformation with Target (QTT)

Description: To check if a transformation is qualified for the target.

QTT (TM, Tran, QS) =

Input:

TM: a target model including selected metrics;

Tran: a single transformation;

QS: qualification score, such as 0.5 by default;

Output:

Boolean: If the Tran is qualified to the target then return true otherwise return false;

Variables:

TID: ID number of the catalog which Tran belongs to;

PM: the number of positive impact;

N: the number of metrics included in TM;

Method:

(1) N := Get_Metrics_number(TM);

(2) PM := Get_Positive_Metrics_number(TM, Tran);

(3) if (PM/N > QS) return true

(4) else return false;

(5) END.

6.5 Summary

The chapter presents the algorithms of program transformation prediction. The evaluation of the transformation impacts and measurement of the reengineering targets are formularised. To recap, the following techniques are used in the proposed algorithms.

- The transformation prediction algorithms are developed based on the relations between target, metric and transformation. The relations are modelled in MOTMET introduced in Chapter 3.

- To predict the transformation steps for a given target is to construct the transformation process model which includes the desired transformation path.

- The transformation process model is expended according to the heuristics, which are addressed in three different scenarios, such as metrics based heuristics, expertise incorporated heuristics and domain features related heuristics.

- The solutions generated from the transformation process are ranked according to the target scores of each transformation path. Software engineer is the one to determine the best solution from the ranked result.

- The algorithm based on metrics without incorporating expertise is much less efficient due to the time complexity of the algorithm. However, it is still a basic algorithm for the scenario where no expertise can be used.

Chapter 7
Tool Support and Case Studies

Objectives

- To illustrate the toolset which supports the proposed approach
- To describe the architecture of the toolset and show the implementation of the toolset
- To give a case study for procedural programming and evaluate the approach by comparing the two strategies, i.e. the one which utilises the proposed approach and another one which does not
- To give a case study for object-oriented program
- To give a case study for multimedia application

7.1 Introduction

For predicting the transformation steps to achieve the reengineering targets, tool support is essential. This chapter introduces a set of the prototype tools, which were developed to provide help with program transformation, target modelling and transformation prediction. Furthermore, the chapter presents three case studies related

to the assessment of the transformation prediction based reengineering approach introduced in this thesis. The case studies will demonstrate the experiments of the proposed approach for procedural program, object-oriented program and multimedia program.

7.2 An Integration Platform

The toolset for the proposed approach is called FermaT Transformation Predictor (F-TP) which acts as an application plug-in integrated in the integration platform called FermaT Integrated Platform (FIP) [21], which extends FermaT Transformation Engine. FIP is developed by the teamwork of Software Technology Research Laboratory at De Montfort University.

7.2.1 Platform Architecture

FIP is a Java based extensible platform for software reengineering with a plug-in mechanism, which provides a number of tools that dedicate to program transformation and comprehension. Figure 7-1 shows the general system architecture of FIP, expressed in three layers: Repository, Core System and Application Plug-ins. F-TP is the plug-in which is developed for the proposed approach in the thesis.

- The *Repository* provides a central place to store and maintain source code and generated data. The data include the information in different representations and in multiple abstract views at various levels stored in the repository.

- The *Core System* provides essential functionalities, including (1) *Transformation Engine*, which provides the program transformation functionalities, (2) *Kernel Runtime*, which provides the plug-in management

and communication functionalities, (3) *Visualisation Engine*, which provides easy-to-use API to create and present diagram and (4) *Repository Access* functionalities are use to retrieve and store the information from the repository.

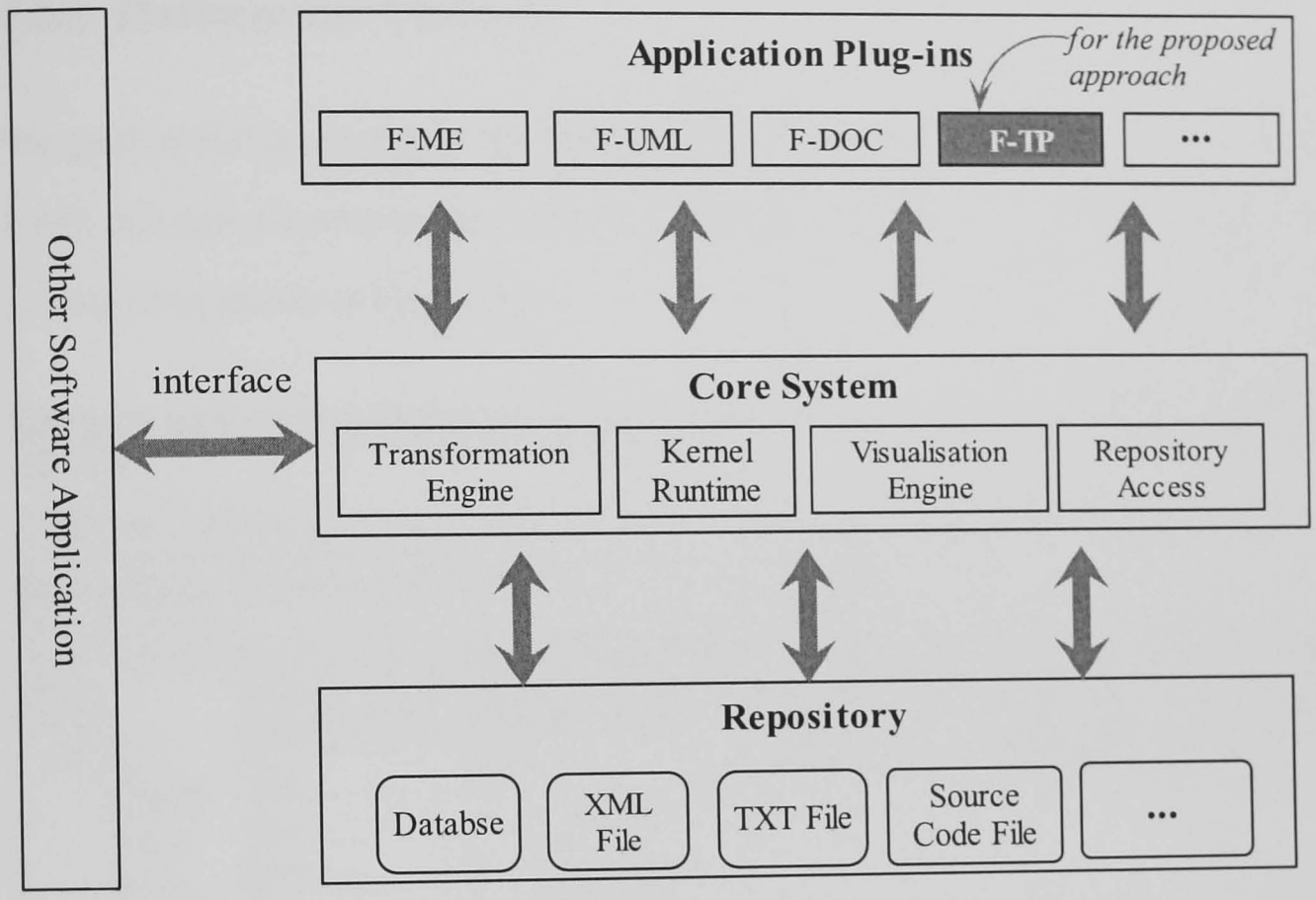

Figure 7-1 FIP Architecture

- The *Application Plug-ins* are a set of tools, providing user interfaces, visualisation and analysis functionalities, for the end-users. FIP UML (F-UML) tool provides the function to extract UML diagram from the legacy system. FIP-Maintainers Environment (F-ME) tool provides an interface for viewing source code and their AST and a transformation handler for applying transformation on selected part of a WSL program. FIP-Documentation (F-DOC) tool is used to extract documentation from WSL source code. FIP-Transformation Predictor (F-TP) is the tool to support the transformation prediction and target modelling. The proposed approach is implemented in FTP by incorporating the other toolset of FIP.

As an integrated reengineering platform, FIP provides the interface to the other software application. The interface includes the translators between the other languages and WSL and the data translation modules.

7.2.2 Platform Environment

The goal of FIP is to integrate the individual tools such as F-UML, F-DOC, F-TP and F-ME into one coherent toolset. To accomplish such a goal, the FIP environment was developed as shown in Figure 7-2.

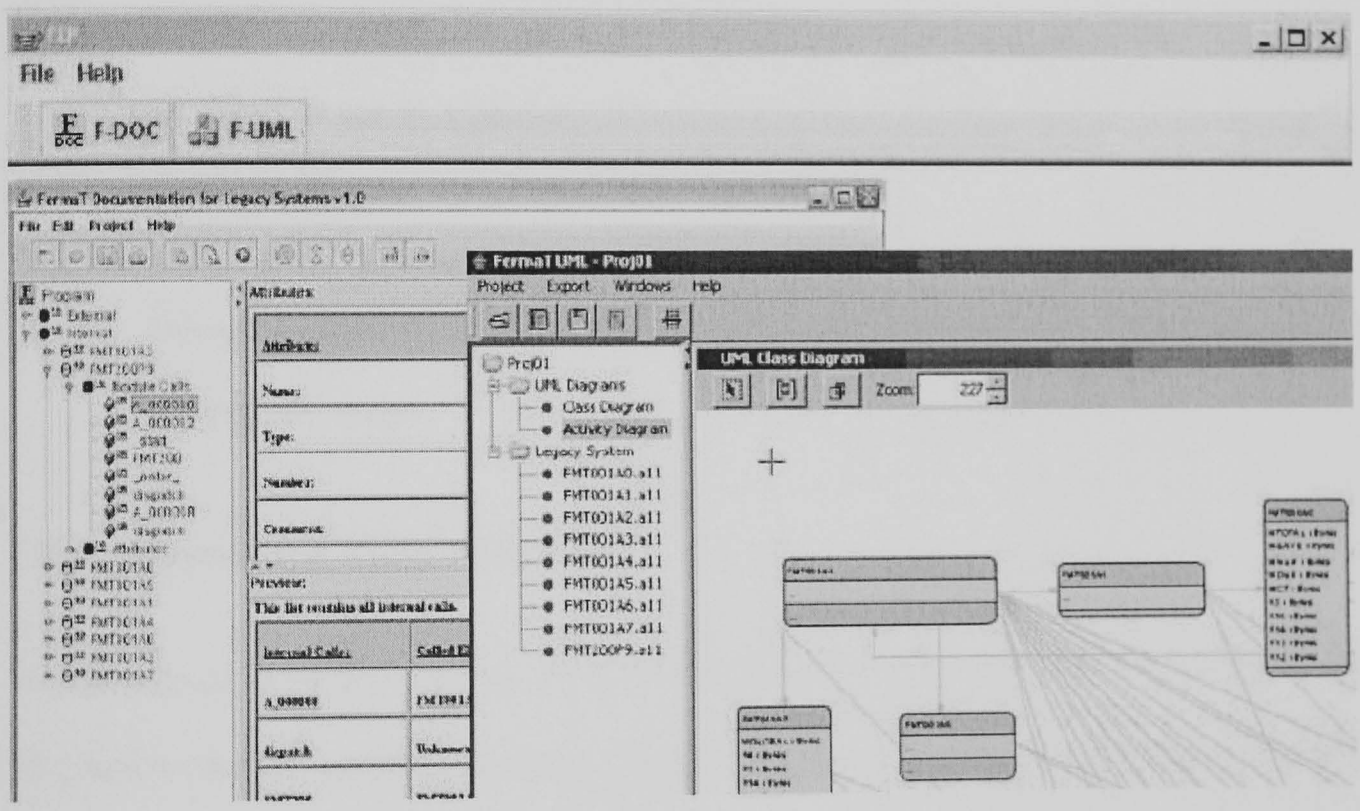

Figure 7-2 FIP Environment

The FIP environment provides the plug-in mechanism by which *Application Plug-ins* can be integrated into the prototype toolset. The FIP environment supports multi-users in distributed environment, which is implemented using Java RMI (Remote Method Invocation). FIP environment can help the maintainers with implement the reengineering process.

7.3 FermaT Transformation Predictor

As an application plug-in of FIP, the FermaT Transformation Predictor (F-TP) is composed of the following interfaces which incorporate the other tools of the integration platform.

- Parser for the WSL extension
- Target modeller to model a target by constructing the sub-target relations and selecting the metrics related to the target
- Metrics viewer to visualise the change of the selected metrics after applying transformations
- Transformation predictor module to elicit the predicted transformations for the determined target

7.3.1 Parser for WSL extension

As the extension of WSL is applied in the proposed approach, the parser of WSL needs to adapt to the change of the language augmentation. For this purpose, Java Compiler Compiler (JavaCC) [58] is used. JavaCC is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognise matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building, actions, debugging, etc.

With JavaCC, the language extension is easier without considering the parser implementation. Figure 7-3 shows that the language designer just needs focus on the

language definition itself. The parser and AST can be generated automatically.

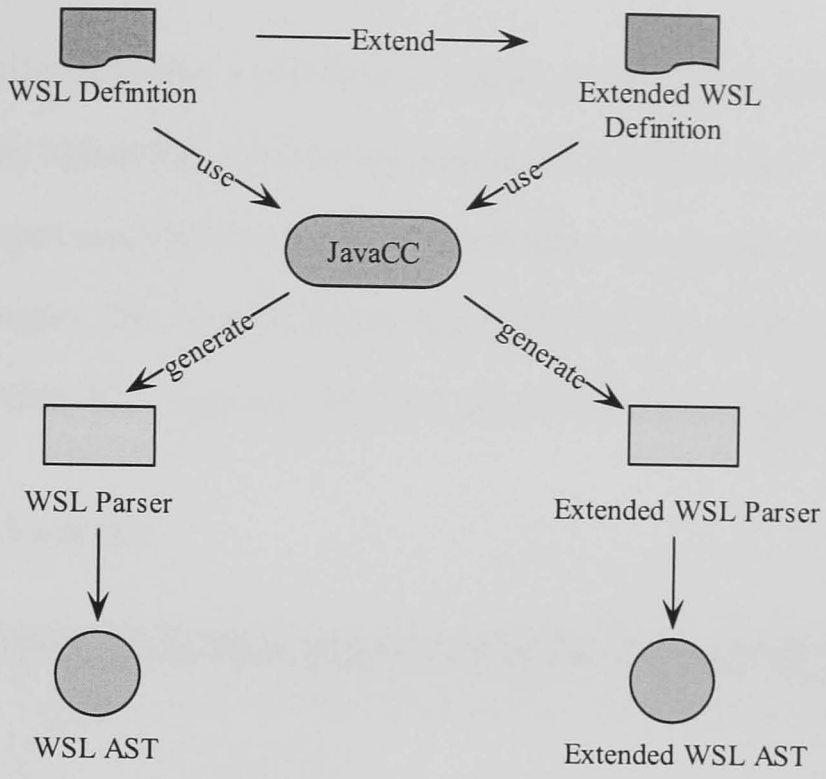

Figure 7-3 Parser Implementation

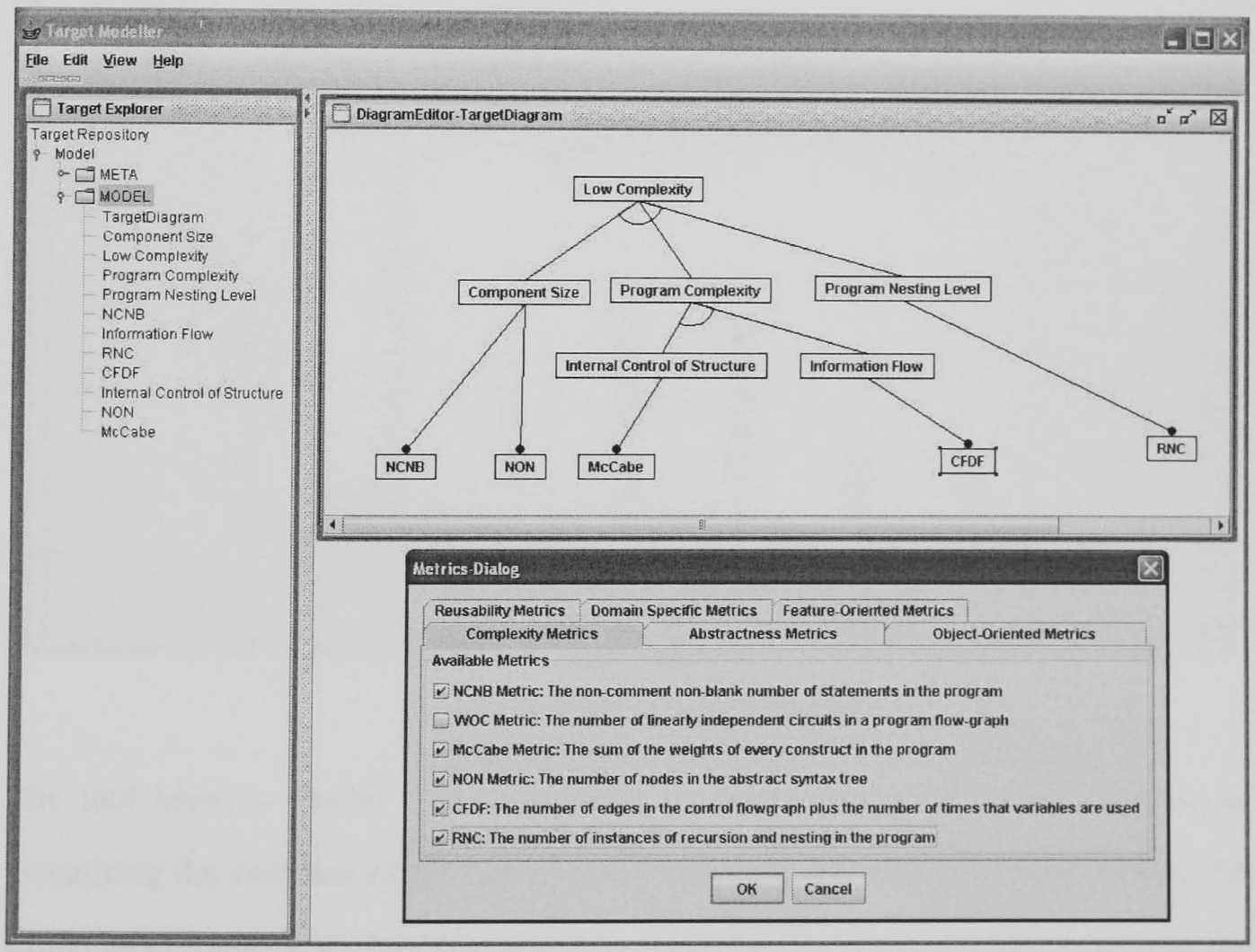

Figure 7-4 Target Modeller Interface

7.3.2 Target Modeller

The Target Modeller provides a platform to construct the target model and attach the relevant metrics by appending them as the leaf nodes on the model. In Figure 7-4, the interface of the target modeller is given. In the interface, the target 'Low Complexity' is chosen as an example. The constructed model is the F-TP is represented as a diagram and stored in an XML file. Appendix B gives an example of a target model in XML.

7.3.3 Metrics Viewer

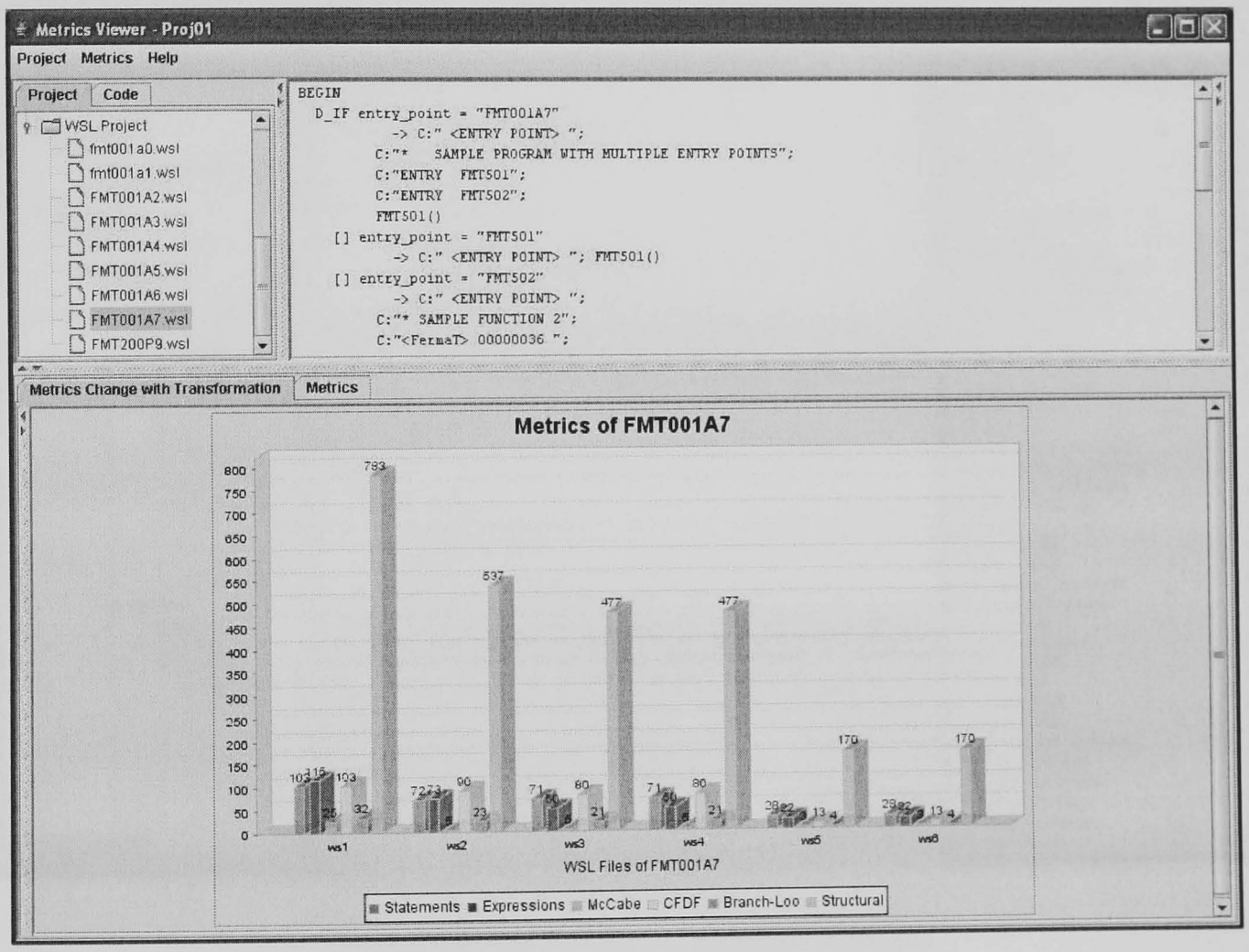

Figure 7-5 Metrics Viewer Interface

The tool Metrics Viewer (MV) contained in the toolset provides the function to visualising the variation of the metrics while applying the transformations. Figure 7-5 shows an example of the interface which includes the tree view of the WSL project, the source code view of the transformed WSL code and the metrics changing view over the

transformation process. The view provides an effective means to the maintainer that how the transformations applied affect the source code valued by the selected metrics.

7.3.4 Transformation Predictor

The Transformation Predictor displays the views of source code, a list of transformations and the prediction result for the determined target, where, the following functions are implemented. Its interface is shown in Figure 7-6.

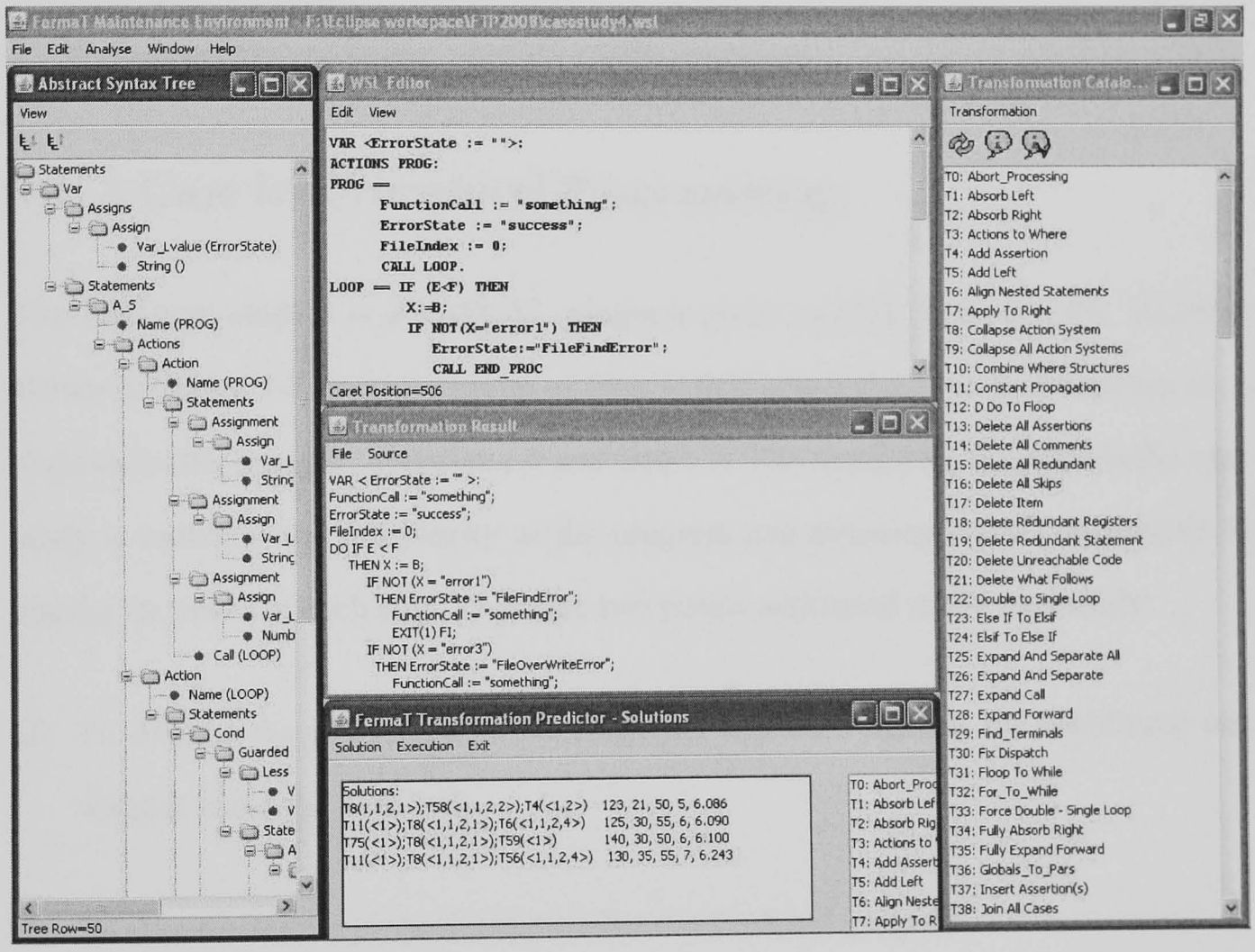

Figure 7-6 Transformation Predictor Interface

- To determine if the expertise incorporated algorithm is eligible for the WSL program;

- To retrieve all of the applicable transformations, whose qualification scores are greater than the threshold, for determined nodes on AST of a WSL program;

- To calculate the target score of a program according to the selected metrics modelled in the Target Modeller;

- To construct the tree structured program transformation model including the transformation paths whose step number is predefined;

- To elicit the predicted result which contains a list of ranked transformation steps;

- To apply the chosen transformation steps and show the result on which the transformation prediction analysis can be continued.

7.4 A Case for Procedural Programming

The first case studied is a PASCAL program given in [75]. The program, which is shown in Figure 7-7, purges a group of files. It first gets a group of files to purge, then finds each file, opens it, overwrites it and erases it. The reengineering target in the case study is reducing the complexity of the program and avoiding GOTO statements. It checks for errors at each step. There are two points addressed in the case study:

(1) To discuss the advantage of the proposed approach against the traditional one without using the approach

(2) To experiment the proposed approach on procedural program

```
1   PROCEDURE PurgeFiles( var ErrorState: ERROR_CODE );
2   var
3       FileIndex:        Integer;
4       FileHandle:       FILEHANDLE_T;
5       FileList:         FILELIST_T;
6       NumFilesToPurge:  Integer;
7   label
8       END_PROC;
9   begin
10      MakePurgeFileList( FileList, NumFilesToPurge );
11      ErrorState := Success;
```

```
12          FileIndex    := 0;
13          while ( FileIndex < NumFilesToPurge ) do
14                begin
15                   FileIndex := FileIndex + 1;
16                   if not FindFile( FileList[ FileIndex ], FileHandle ) then
17                   begin
18                         ErrorState := FileFindError;
19                         goto END_PROC
20                   end;
21                   if not OpenFile( FileHandle ) then
22                   begin
23                         ErrorState := FileOpenError;
24                         goto END_PROC
25                   end;
26                   if not OverwriteFile( FileHandle ) then
27                         begin
28                         ErrorState := FileOverwriteError;
29                         goto END_PROC
30                   end;
31                   if Erase( FileHandle ) then
32                         begin
33                         ErrorState := FileEraseError;
34                         goto END_PROC
35                   end
36                end; { while }
37          END_PROC:
38                DeletePurgeFileList( FileList, NumFilesToPurge )
39    end;
```

Figure 7-7 A PASCAL Program

7.4.1 Strategy without Using Transformation Prediction Approach

Without using the proposed transformation prediction method, the user needs to keep the two aspects of the target in mind. In order to eliminate the GOTO statement, a standard, textbook, structured-programming approach is to rewrite with nested **if** statements, nest the **if** statements so that each is executed only if the previous test succeeds.

By this standard approach, the transformation Reverse_if, which reverses the two arms of a simple **if** statement, can be adopted to nest the **if** statements. The transformation needs to be applied for the **if** statements at Line 16, 21, 26, 31 respectively. The process merely follows the guidance of the structured-programming. The result of the GOTO removing can be acquired by performing the transformation for 4 times and shown in Figure 7-8. The manual determination of the three factors, i.e., the transformation, the

position where the transformation is applied and the transformation execution steps, can result in the program without GOTOs as shown in Figure 7-8.

```
1   PROCEDURE PurgeFiles( var ErrorState: ERROR_CODE );
2   var
3         FileIndex:          Integer;
4         FileHandle:         FILEHANDLE_T;
5         FileList:           FILELIST_T;
6         NumFilesToPurge:    Integer;
7   begin
8      MakePurgeFileList( FileList, NumFilesToPurge );
9      ErrorState   := Success;
10     FileIndex    := 0;
11     while ( FileIndex < NumFilesToPurge and ErrorState = Success ) do
12        begin
13           FileIndex := FileIndex + 1;
14           if FindFile( FileList[ FileIndex ], FileHandle ) then
15           begin
16              if OpenFile( FileHandle ) then
17              begin
18                 if OverwriteFile( FileHandle ) then
19                 begin
20                    if not Erase( FileHandle ) then
21                    begin
22                       ErrorState := FileEraseError
23                    end
24                 end
25                 else
26                 begin
27                    ErrorState := FileOverwriteError
28                 end
29              end
30              else
31                 begin
32                    ErrorState := FileOpenError
33                 end
34           end
35           else
36           begin
37              ErrorState := FileFindError
38           end
39        end; { while }
40     DeletePurgeFileList( FileList, NumFilesToPurge )
41  end;
```

Figure 7-8 A Result by the Strategy without Using the Proposed Approach

However, it is observed that the result neglects another aspect of the target, i.e., the deep nesting level raises the complexity of the program. With nesting like this, to understand the code, the user has to keep the whole set of nested **if**s in his/her mind at once. Moreover, the distance between error-processing code and code that invokes it is too far: the code that sets ErrorState to FileFindError, for example, is Line 22 from the

if statement that invokes it. On the other hand, it is hard to assess the process and the result without quantitative control although the feature of the result that the GOTOs have been removed is obvious.

In short, the target is not satisficed by the traditional approach without using the target driven transformation prediction. Next section will discuss the effect of using the proposed approach.

7.4.2 Strategy with Target Driven Transformation Prediction

The proposed transformation prediction approach follows three main steps, namely (1) modelling the target; (2) constructing the transformation process model; and (3) generating transformation paths as solutions. The target model chosen is the 'Low Complexity (LC)' model and several metrics are selected as constrain of the prediction approach. The model is depicted as Figure 7-9. In this model, the elimination of GOTOs is not included obviously, because this sub-target can be achieved by selection of predicted solution whose transformation result does not have GOTOs.

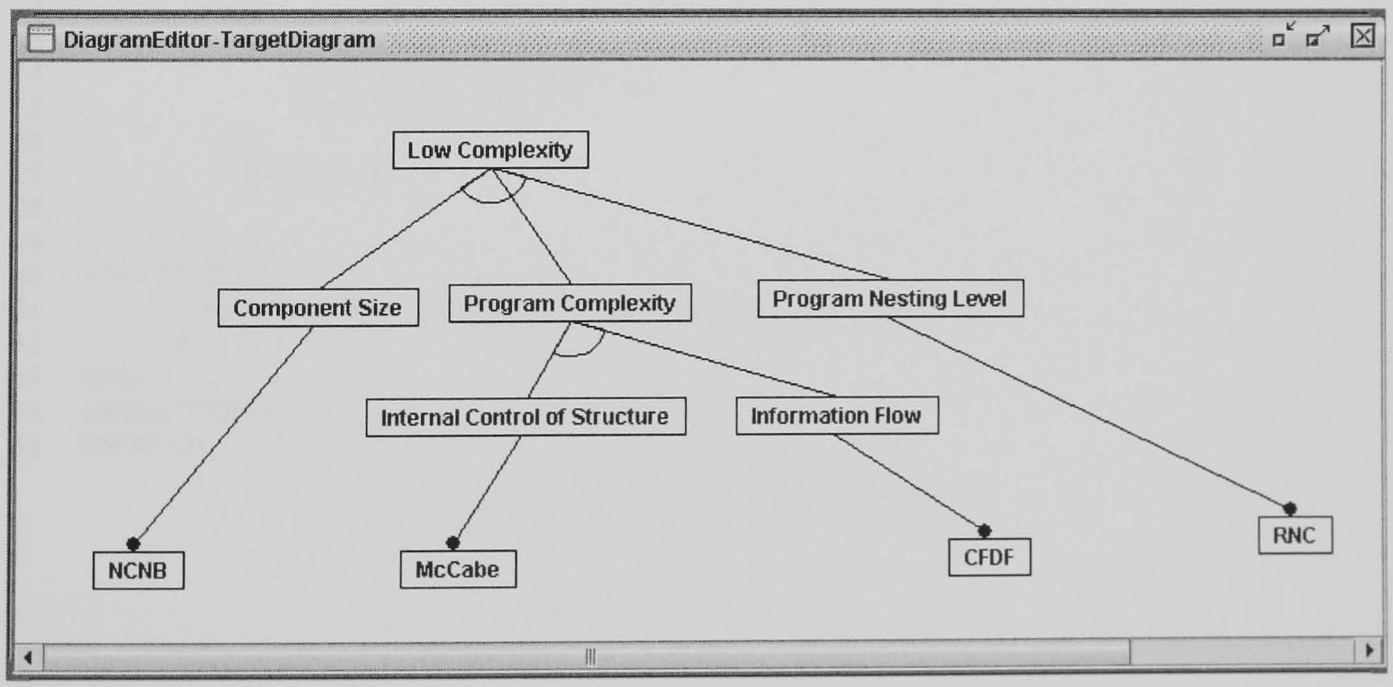

Figure 7-9 'Low Complexity' Target Model

Before performing the transformation, the PASCAL source code is translated to WSL program. In order to translate the procedural program which has GOTO statements shown in Figure 7-7 to WSL, all the labels need to be at the top level, so that they can be converted to actions in an action system. This is easily accomplished by implementing the **while** loop as action *LOOP* at the top of the top of the loop and a **call** *LOOP* at the end of the loop. The translated WSL program is shown in Figure 7-10. Figure 7-11 displays the Abstract Syntax Tree (AST) generated by FME [21].

```
1    VAR <ErrorState := "">:
2    ACTIONS PROG:
3    PROG ==
4          <FileIndex := 0; NumFilesToPurge := 0>;
5          !P MakePurgeFileList(FileList,NumFilesToPurge);
6          ErrorState := "Success";
7          FileIndex := 0;
8          CALL LOOP.
     LOOP ==
9          IF (FileIndex < NumFilesToPurge) THEN
10               FileIndex := FileIndex + 1;
11               IF NOT !P FindFile(FileList[FileIndex],FileHandle) THEN
12                     ErrorState := "FileFindError";
13                     CALL END_PROC
14               FI;
15               IF NOT !P OpenFile(FileHandle) THEN
16                     ErrorState := "FileOpenError";
17                     CALL END_PROC
18               FI;
19               IF NOT !P OverwriteFile(FileHandle) THEN
20                     ErrorState := "FileOverwriteError";
21                     CALL END_PROC
22               FI;
23               IF !P Erase(FileHandle) THEN
24                     ErrorState := "FileEraseError";
25                     CALL END_PROC
26               FI;
27               CALL LOOP
28         FI
29         CALL Z.
30   END_PROC==
31         !P DeletePurgeFileList( FileList, NumFilesToPurge );
32         CALL Z
33   END
34   ENDACTIONS
35   ENDVAR
```

Figure 7-10 Translated WSL Program

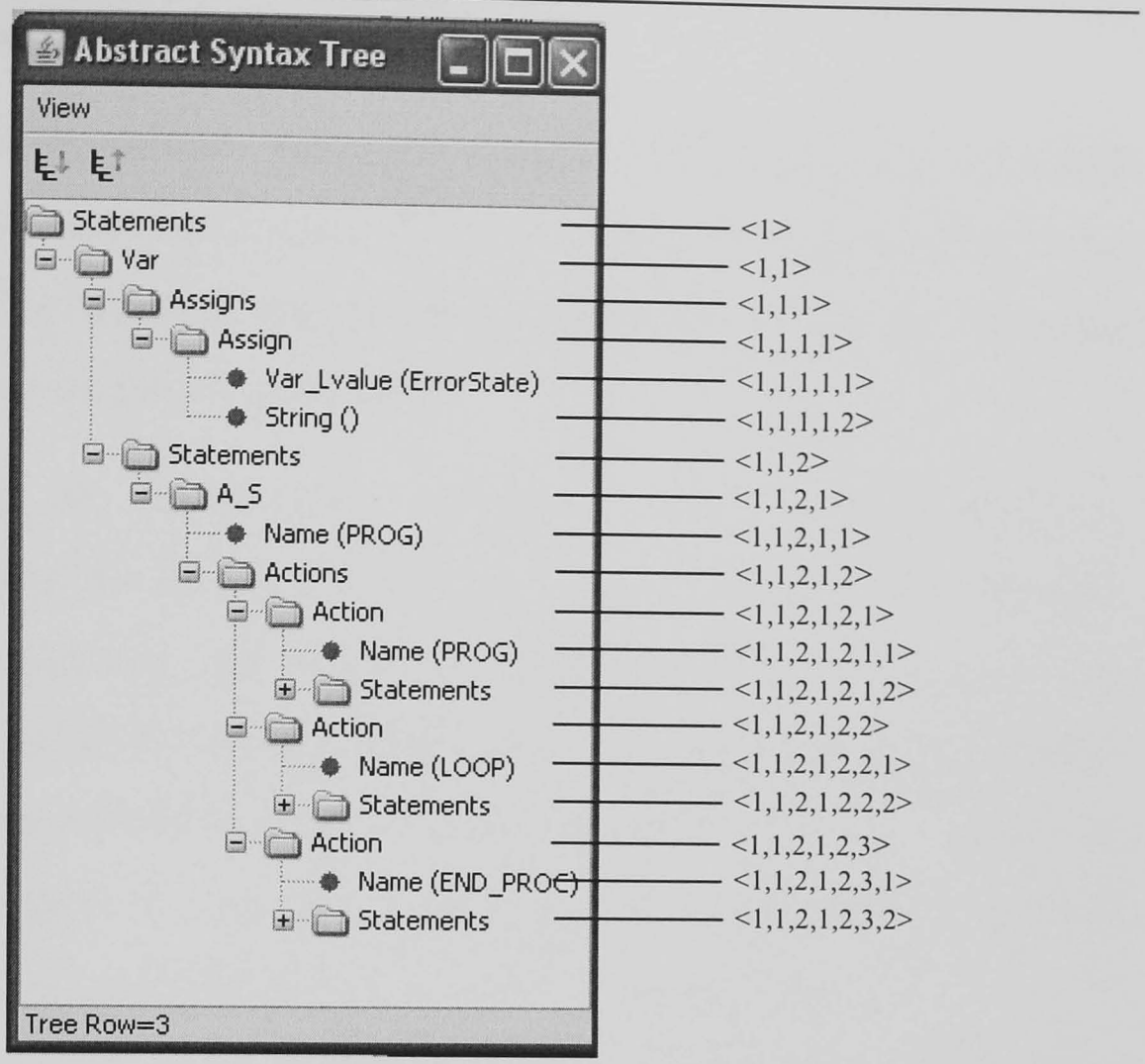

Figure 7-11 A View of the WSL Program AST

In the target model, the target 'Low Complexity' can be positively satisficed by the four selected metrics. Therefore, the target score representing the impact of a transformation on the program can be evaluated by Formula 6-1 and Formula 6-2. The impact function is formulated as $\frac{Benefit}{Cost}$. In the case study, the impact can be evaluated as the following formula.

$$impact_p(t_i) = \frac{-\left(\frac{(value_p(NCFC)-29)}{\sqrt{(value_p(NCFC-29)^2+29}} + \frac{(value_p(McCabe-22)}{\sqrt{(value_p(McCabe)^2+22}} + \frac{(value_p(CFDF)-13)}{\sqrt{(value_p(CFDF-29)^2+13}} + \frac{(value_p(RNC)-1)}{\sqrt{(value_p(RNC-1)^2+1}}\right)}{\frac{\#added\_nodes+\#modified\_nodes+\#deleted\_nodes}{\sqrt{(\#added\_nodes+\#modified\_nodes+\#deleted\_nodes)^2+100}}}$$

By using the metrics based prediction approach, the transformation predictor will construct the transformation process model which represents the state transition of the program. The process starts at the AST node at the position <1>, i.e., the *Statements* node. The results at the first step are the transformations, which can be applied on the

whole WSL program.

According to the algorithm to construct the transformation process model, if the target score of a transformation is equal to 0, i.e., the program is not changed or affected after applying the transformation, then the transformation will not have any succeeding vertex in the transformation process model.

In this case, the transformation candidates include *Constant_Propagation*, *Delete_All_Redundant*, *Remove_All_Redundant_Vars* and *Simplify* whose condition functions are matched the state functions of the program and qualification scores are greater than threshold 0.5. However, after testing the four transformations, the values of the metrics are still same as before. The result is shown in Table 7-1. Therefore, the four transformations are not taken into account for the identified target 'Low Complexity' for the case although they can be used for the program.

Transformation \ Metrics	NCNB	McCabe	CFDF	RNC	Impact
Before Transformation	29	13	22	1	—
Constant_Propagation (<1>)	29	13	22	1	0
Delete_All_Redundant (<1>)	29	13	22	1	0
Remove_All_Redundant_Vars (<1>)	29	13	22	1	0
Simplify (<1>)	29	13	22	1	0

Table 7-1 Impact of the Transformation on Node <1>

With the failure of finding the transformations at the position <1> on the AST, the prediction algorithm will go down to the next level on the tree and construct the search graph for the nodes at the second level as well as the succeeding levels. Before running the algorithm, the parameter *Iteration* is initialised as 11, i.e., the prediction algorithm for this case will dig over the transformations for the nodes on the AST up to 11 levels.

The algorithm cannot find any positive result since no metrics are affected until the algorithm reaches the node at the position <1, 1, 2, 1>. For this node whose specific

type is A_S, after testing the available transformations, the code can be transformed by being affected with the selected features which measured by the metrics. The impact of the transformations is calculated in Table 7-2. Meanwhile, the value of the heuristic can be obtained according the defined heuristic function. Herein, T1: Collapse_Action_System belongs to the Rewrite group which has the qualification score 1 to the target 'Low Complexity', while T3: Merge_Calls belongs to the Simplify group which has the qualification score 1 to the target. The transformations which do not change the source code features will not be taken into account.

Transformation \ Metric	Cost	Benefit				Impact
		NCNB	McCabe	CFDF	RNC	
Before Transformation	—	29	13	22	1	—
T8 (P1<1,1,2,1>)	0.832	22	6	22	2	1.172
T15 (P1<1,1,2,1>)	0.734	29	13	22	1	0
T41 (P1<1,1,2,1>)	0.625	26	8	22	1	2.076
T68 (P1<1,1,2,1>)	0.563	29	13	22	1	0
T75 (P1<1,1,2,1>)	0.732	29	13	22	1	0
T31 (P2<1,1,2,4>)	0.894	25	6	20	2	1.412
T56 (P2<1,1,2,4>)	0.447	27	7	21	2	1.262
T31 (P3<1,1,2,4>)	0.813	26	7	23	2	0.970
T75 (P3<1,1,2,4>)	0.707	27	7	20	2	1.374
T50 (P4<1,1,1,1,2,1,2>)	0.514	22	7	20	4	1.245
T50 (P5<1,1,1,1,1,2,1,2>)	0.447	20	7	20	5	1.209
T50 (P6<1,1,2,2,1,1,1,2,2,1,1>)	0.371	24	6	22	3	1.221
T50 (P7<1,1,2,2,1,1,1,2,2,1,2>)	0.371	23	6	22	4	1.220
T8 (P8<1,1,2,1>)	0.707	20	7	23	5	1.561
T8 (P6<1,1,2,1>)	0.748	25	7	22	2	1.439
T56 (P9<1,1,2,4,1,1>)	0.514	26	7	22	2	1.314
T84(P9<1,1,2,4,1,1>)	0.371	24	6	20	1	1.125

Table 7-2 Impact of the Selected Transformations on the Case Study 1

At this step, the transformation process model can be generated as follows. In this case, the constructed model only contains the sequence relations between transformations because there is no effective transformation which can be applied for the parallel statements. The tree is constructed by taking both cost and benefit into account.

After applying a transformation, the source code could be altered so that the nodes at

the same position are changed as well. Therefore, the predictor has to search the transformations applied from the node at the position <1>. Any transformation, whose impact is 0, is not added into the graph. The graph in Figure 7-12 is only for the transformation process containing three steps. After finding the non-dominated transformation sequence, the software engineer can continue the prediction process based on the current state of the program after applying the predicted transformations.

From the transformation process model, the transformation paths can be ranked as the predicted steps, which are the potential solutions for the given problem.

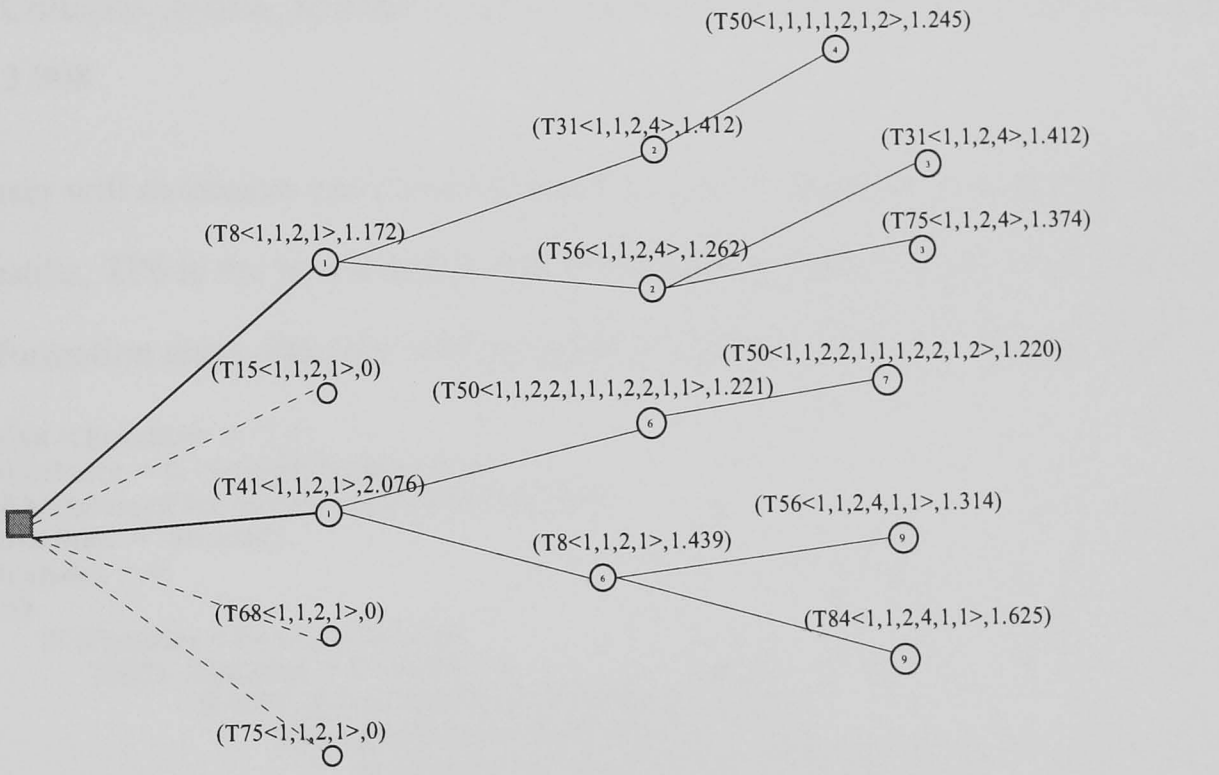

Figure 7-12 Constructed Transformation Process Model

TP6: Merge_Calls<1,1,2,1> ∧ Collapse_Action_System<1,1,2,1> ∧
Double_to_Single_Loop<1,1,2,4>
TS6: 5.140

TP5: Merge_Calls<1,1,2,1> ∧ Collapse_Action_System<1,1,2,1> ∧
Reduce_Loop<1,1,2,4,1,1>
TS5: 4.829

TP4: Merge_Calls<1,1,2,1> ∧ Move_To_Right<1,1,2,2,1,1,1,2,2,1,1> ∧
Move_To_Right<1,1,2,2,1,1,2,2,1,1>

TS4: 4.517

TP2:Collapse_Action_System<1,1,21>∧Reduce_Loop<1,1,2,3>∧
Floop_To_While<1,1,2,4>

TS2: 3.846

TP1: Collapse_Action_System<1,1,21> ∧ Floop_To_While<1,1,2,4>
∧ Move_To_Right<1,1,1,1,2,1,2>

TS1: 3.829

TP3: Collapse_Action_System<1,1,21>∧Reduce_Loop<1,1,2,3>∧Simplify<1,1,2,4>

TS3: 3.808

The user will determine which transformation path to apply the transformations. From the results, TP6 is the best solution due to the highest target scores. After applying the transformation steps, the new WSL program is altered as shown in Figure 7-13.

```
1   VAR < ErrorState := "" >:
2   <FileIndex := 0; NumFilesToPurge := 0>;
3   !P MakePurgeFileList(FileList,NumFilesToPurge);
4   ErrorState := "Success";
5   FileIndex := 0;
6   DO
7       IF (FileIndex < NumFilesToPurge)
8           THEN  FileIndex := FileIndex + 1;
9                 IF NOT !P FindFile(FileList[FileIndex],FileHandle)
10                      THEN ErrorState := "FileFindError";
11                           !P DeletePurgeFileList( FileList, NumFilesToPurge );
12                           EXIT(1)
13                ELSIF NOT !P OpenFile(FileHandle)
14                      THEN ErrorState := "FileOverWriteError";
15                           !P DeletePurgeFileList( FileList, NumFilesToPurge );
16                           EXIT(1)
17                ELSIF NOT !P OverwriteFile(FileHandle)
18                      THEN ErrorState := "FileOpenError";
19                           !P DeletePurgeFileList( FileList, NumFilesToPurge );
20                           EXIT(1)
21                ELSIF !P Erase(FileHandle)
22                      THEN ErrorState := "FileEraseError";
23                           !P DeletePurgeFileList( FileList, NumFilesToPurge );
24                           EXIT(1)
25                FI
26      ELSE EXIT(1)
27      FI
28  OD
29  ENDVAR
```

Figure 7-13 Transformed WSL Program by Applying TP6

The result from the TP6 does not contain any GOTOs and nesting level of **if** statements is only 2. It does not have the problem that the first strategy caused. Besides, the variation of the metric value shows the quality w.r.t complexity is improved.

The other sequences contained in the model are also elicited to the software engineer who makes the decision which sequence is chosen to apply. After applying the selected transformation sequence, the transformed program will be regarded as a new start on which the software engineer can carry on the further reengineering analysis.

7.4.3 Comparison of Two Strategies

By comparing the results from the two different strategies, the following conclusions can be gained.

- Using the proposed approach, the correctness of the result, which is required to satisfice the given reengineering target, can be improved. The first strategy to remove the GOTO statements resulted in the deepest nesting level as 6. The result increases the complexity rather than decreases it. However, the deepest nesting level is only 3. In terms of the result about the nesting level, the second one has a better result. The strategy with the proposed approach can give attention to both aspects of the target.

- The first strategy lacks of the quantitative means to control the process so that it is hard to evaluate the target satisficed degree. While the second one is guided by the quantitative approach towards the desired target. It is easier to compare the potential solutions and predict the needed one by the latter strategy.

- Although in this case studied by using the second strategy the expertise the expertise incorporated algorithm is not applied, it is obvious that in some case if an expertise rule is eligible for the proposed problem and it is applied, the correctness and efficiency can be ensured against the situation where the user selects and applies transformations without the help of the expertise.

7.5 A Case for Object Oriented Program

As for the object oriented case, a bioinformatics application Linkage Disequilibrium Analyser (LDA) [35], which was published in the *Bioinformatics* journal, is selected to for this study. LDA is an integrated java-based program that provides elaborate graphic and plain-text output of pair-wise linkage disequilibrium analysis of single nucleotide polymorphism genotypic data. It takes a simple flat-file as input, provides a dialogue to set up parameters and for optional selection of the different test method and presents the analysis results both in the form of graphics and plain-text. Figure 7-14 shows a screenshot of the application.

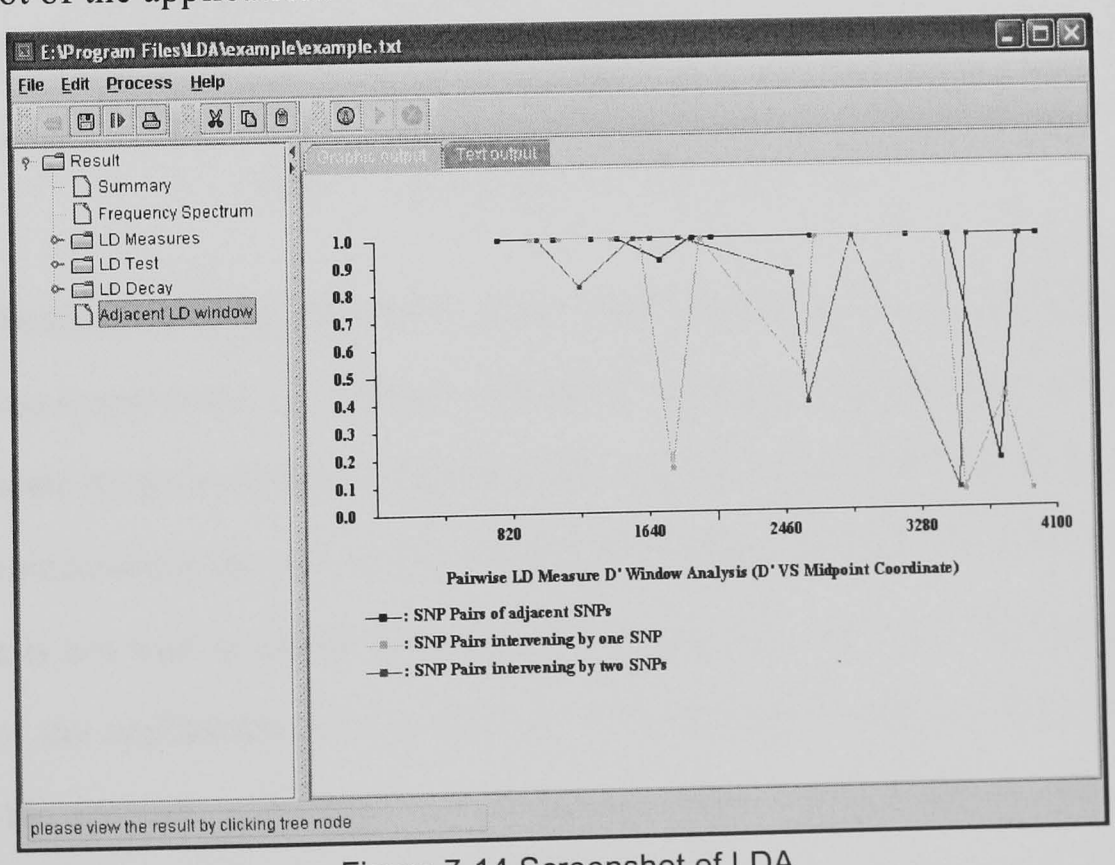

Figure 7-14 Screenshot of LDA

Metrics	Definition
AMS	Average Module Size: This measure is calculated by number of statements over number of methods. $AMS = \frac{LOC}{Number\ of\ methods}$
ANFC	Average number of features implemented in a class
NCIF	Average Number of classes which implement a feature
OSC	Overlap statements among the classes

Table 7-3 Selected Metrics for the LDA Analysis

Feature	Classes	Feature	Classes
Open	LoadFile	***Summary***	NucleotideDiversity
Save	FileUtility	***FS***	FreqSpec
Print	JComponentVista Vista	***FS Graphic***	FreqSpec
Cut	CutTextAction	***LD Measures***	LdMeasure_EM
Copy	CopyTextAction	***LD Measure Graphics***	LdmGraphic
Paste	PasteTextAction	***LD Test***	LdTextOutput
Main GUI Event	LDAaPP OptionsSettingsTree_mouseAdapter	***LD Test Graphics***	LddGraphic
LD Decay	LdDecay	***LD Decay Graphic***	LdDecayGraphic
Main GUI	LDAaPP, LDTree, OptionsSeeting_TreeNodes		
Option	OptionsDialog_hwechichk_changeAdapter OptionsDialog_hwechk_changeAdapter OptionsDialog_ldtchichk_changeAdapter OptionsDialog_ldtchk_changeAdapter OptionsDialog		

Table 7-4 Feature-Class Table of the LDA

The application has about 5000 lines of code and 2 packages which contain 29 classes and 1 class respectively. To choose it as a case study relies on three reasons: (1) it is a normal scale program which is suitable for the transformation experiment; (2) it is used by the bioinformaticians in practice; (3) After analysing the source code, it is found that the code is not well structured and the modules are high coupled. The reengineering target for the application is to extract the feature-oriented modules. To evaluate the state of the program with respect to the reengineering target, in the target model, the metrics shown in Table 7-4 are selected from the object-oriented, reusability and

feature oriented metrics group. In the target model, they are correlated by AND relations and negatively contribute the target.

Before performing the transformation prediction process, the feature-oriented analysis is necessary for the determined target. In the stage, the main features of the application and the relations between the features and the classes are captured. Table 7-4 lists the main features of LDA and the implementation of those features. In the analysis, it can be examined that there are two problems, which may not help the maintenance and reuse of the application.

Transformation \ Metric	Cost	Benefit				Impact
		AMS	ANFC	NCIF	OSC	
Before Transformation	—	163	4	3	20	—
T91 (P1, C1, <posn1>) /\ T91 (P1, C2, <posn2>) /\ T91 (P1, C3, <posn3>)	0.234	145	4	3	20	0.432
T39 (P1, C1, <posn1>) /\ T39 (P1, C2, <posn2>) /\ T39 (P1, C3, <posn3>)	0.342	145	4	3	20	0.342
T79 (P2, C1, <posn4>) /\ T79 (P2, C2, <posn5>) /\ T79 (P2, C3, <posn6>)	0.622	140	3.6	2.6	20	0.234
T93 (P2, C1, <posn7>) /\ T93 (P2, C2, <posn8>) /\ T93(P2, C3, <posn9>)	0.434	143	3.6	2.5	14	0.563
T100 (P3, Drawgraph)	0.234	132	3.6	2.4	10	0.244
T79 (P4, C1, <posn10>) /\ T79 (P4, C2, <posn11>) /\ T79 (P4, C3, <posn12>)	0.523	145	3.6	2.4	10	0.256

T91: Movement; T93: Wrapper; T100: Make_Class
T39: Make_Procedure; T50: Move_to_Left; T66: Rename_Proc;
T79: Substitute_and_Delete; T82: Take_Out_Left; T83: Take_Out_Right

Table 7-5 Impact of the Selected Transformations on the Case Study 2

- The classes for 'Main GUI' are not well structured. The class 'LDAaPP' to draw the main frame has 1510 lines of code and contained the code to draw the frame and write the event. It is hard to read and locate the maintenance

needs. The code of the presentation and the event are so coupling that could cause more problems during the maintenance.

- The statistic graph generation feature is cross-cutting in the data analysis modules which have their own functions to draw the graphs. The three classes 'LdDecayGraphic', 'LdmGraphic', 'LddGraphic' have the overlap part to initialise the graphic environment, render the graph and process the parameters. This overlap can cause the problem that software engineer has to modify the code in the three class if the graph initialisation feature.

In the case study, the second problem is investigated and given the solution based on the proposed approach. The transformation prediction algorithm is used for the feature-oriented class extraction and generation. The selected metrics will guide the prediction process. The transformations are selected according to their condition functions and the state functions of the classes.

$P_i, (i = 1,2,\cdots)$: Program version number;

$C_j, (j = 1,2,3,4)$: The three classes and the new feature-oriented class.

The details of the TPM computation is shown in Table 7-5. By performing the prediction algorithm, the transformation process model generated only contains one transformation path as follows.

T91(P1, C1, <posn1>) /\ T91 (P1, C2, <posn2>) /\ T91 (P1, C3, <posn3>);

T93 (P2, C1, <posn7>) /\ T93 (P2, C2, <posn8>) /\ T93 (P2, C3, <posn9>);

T100 (P3, DrawGraph);

T79 (P4, C1, <posn11>) /\ T79 (P4, C1, <posn12>) /\ T79 (P4, C1, <posn13>)

The result is to generate a new class DrawGraph and modified the corresponding code of the other three classes.

```
Class DrawGraph {
    Method DrawGraph (colour, sizearr, opaque ) {
        setbackground (colour);
        setsize (sizearr);
        setOpaque (opaque);
    }
```

The transformed result including the new generated class and the revised old classes can be translated back to Java.

7.6 A Case for Multimedia Program

Multimedia languages are a new class of languages that have arisen in the past decade. The term 'multimedia' refers to "a presentation or display that involves more than one method or medium of presentation"[73]. Such media may include audio, video, still images and animations that accompany the standard text display. Therefore, a multimedia application is one that uses and includes more than one of these media in a cohesive manner. A multimedia language "is a set of software tools for creating multimedia applications" [73]. All multimedia languages present the developer with a set of software tools to aid in the development process.

For example, the Synchronised Multimedia Integration Language (SMIL) is a popular multimedia synchronised language and allows a user to define how independent media objects are to be integrated into a media representation and it provides rules and define when (and if) various media objects are actually rendered for the end-user. Existing popular players for Synchronised Multimedia Integration Language (SMIL) [19] includes Microsoft Internet Explorer, Apple QuickTime, RealNetworks Realplayers,

AMBULANT Open SMIL Player [30] and GriNs [87]. The generator of the presentation normally can be a specific editor, such as GriNs or a normal text editor. However, without these players, it is not easy to understand the plain XML source code and capture the spatial as well as the temporal relations between the media objects.

The case study is performed based on an SMIL application. The reengineering target identified for the multimedia program is to extract the abstraction of the temporal behaviour of the multimedia application. In the other word, the program transformations applied for approaching the target should be able to strip off the other information irrelevant to the temporal properties as much as possible. This target can be obtained through the transformations, which raise the abstraction level of the multimedia presentation.

In Chapter 5, the WSL language and the transformations are extended to accommodate the analysis of object oriented program and multimedia application. It has been concluded that the most distinct characteristics in this area is the data type and operation on the data. Different types of multimedia data have different properties and functions, so that it appears intuitive to describe them by different data types. This approach allows the specification of a class for each data type; normally, this includes classes like text, image, audio and video.

Figure 7-15 is a simple multimedia application written in SMIL.

```
<?xml version="1.0"?>
<!DOCTYPE smil PUBLIC "-//W3C//DTD SMIL 2.0//EN"
                              "http://www.w3.org/2001/SMIL20/SMIL20.dtd">
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
  <head>
    <meta name="title" content="Happy Birthday, Large Screen Version"/>
    <meta name="generator" content="GRiNS Pro for SMIL 2.0, v2.2 Mobile win32 build 151"/>
    <meta name="author" content="Dick Bulterman"/>
```

```
    <layout>
        <root-layout id="Player-Window" backgroundColor="gray" width="380" height="270"/>
        <region id="audio" soundLevel="10%"/>
        <region id="bkgd_image" left="0" width="380" top="0" height="270"/>
        <region id="Video" left="92" width="280" top="6" height="216" z-index="1"/>
        <region id="Captions" left="130" width="220" top="229" height="20" z-index="2"/>
        <region id="Menu" left="2" width="84" top="7" height="260" z-index="1"/>
        <region id="unnamed-region" title="unnamed region" left="0" top="0"/>
        <region id="unnamed-region-1" title="unnamed region" left="0" top="0"/>
    </layout>
    <transition id="slideover" type="slideWipe"/>
    <transition id="fade" type="fade"/>
    <transition id="push" type="pushWipe"/>
</head>
<body>
    <par id="BigBirthday">
        <seq>
            <par id="Intro" endsync="SkipIntro">
                <seq>
                    <img id="FBT" region="Video" begin="5" dur="4s"
                        fill="transition" src="In-1.gif" transIn="fade"/>
                    <img id="FBT-0" region="Video" begin="0.9"
                        dur="3.2s" fill="transition" src="In-2.gif" transIn="fade"/>
                    <img id="FBT-1" region="Video" begin="2" dur="15s" fill="freeze"
                        src="In-3.gif" transIn="fade" transOut="fade"/>
                </seq>
                <audio id="HmGeb" region="unnamed-region-1" src="HappyBirthday.mp3"/>
                <img id="SkipIntro" fill="freeze" end="HmGeb.end;activateEvent"
                    src="Skip.png" region="Menu" top="193" />
</par>
<par id="MenuImages" dur="indefinite">
    <img id="F1s" region="Menu" src="HBdata/F1s.jpg"
        width="81" top="5" height="54" transIn="slideover"/>
    <img id="W1s" region="Menu" src="W1s.jpg" width="80"
        top="66" height="53" transIn="slideover"/>
    <img id="A1s" region="Menu" src="A1s.jpg" width="81"
        top="123" height="54" transIn="slideover"/>
    <img id="FBTs" region="Menu" src="FBTs.jpg" width="81"
        top="195" height="54" transIn="slideover"/>
    <excl id="Videos" dur="indefinite" fillDefault="freeze">
        <par id="Fz3" begin="0; F1s.activateEvent">
            <video id="Fz3-0" region="Video" src="Fz3-g.mpg"/>
            <audio id="Birthday" region="audio" src="Birthday.mp3"/>
        </par>
        <par id="Wz3" begin="W1s.activateEvent">
            <video id="W1s-1" region="Video" src="Wz3-g.mpg"/>
            <audio id="Birthday-0" region="audio" src="Birthday.mp3"/>
        </par>
        <par id="Az3" begin="A1s.activateEvent">
            <video id="Az3-0" region="Video" src="Az3-g.mpg"/>
            <audio id="Birthday-1" region="audio" src="Birthday.mp3"/>
        </par>
        <par begin="FBTs.activateEvent">
            <seq>
                <img id="FBT-n" region="Video" dur="5.9s" fill="transition" src="FBT.jpg"/>
                <video id="BTz3" region="Video" fill="transition"
                    src="BTz3-g.mpg" transIn="fade"/>
                <img id="FBP" region="Video" dur="9.56667s" fill="transition"
```

```
                        src="FBP.jpg" transIn="slideover" transOut="push"/>
                    <video id="BPz3" region="Video" src="BPz3-g.mpg" transIn="fade"/>
                  </seq>
                  <audio id="V0RB0KFJ" region="audio" src="Ballgame.mp3"/>
                </par>
              </excl>
            </par>
          </seq>
          <img id="BkgdImg" region="bkgd_image" fill="freeze" src="Back3s.gif"/>
        </par>
      </body>
</smil>
```

Figure 7-15 An SMIL Multimedia Application

In the process of translation from SMIL to the extended WSL, the translator needs to retrieve the media data from the XML document and construct the media class for those data. At this step, only the constructs and attributes related to the spatial and the temporal features are translated to WSL corresponding constructs. The result of this step is shown as follows.

```
Begin
   regionaudio := HASH_TABLE;
   regionaudio.("soundlevel"):= 0.1;
regionbkgd_image := HASH_TABLE;
regionbkgd_image.("left") := 0; regionbkgd_image.("width") := 380;
regionbkgd_image.("top")   := 0; regionbkgd_image.("height"):= 270;

regionVideo := HASH_TABLE;
regionVideo.("left") := 92; regionVideo.("width") := 220;
regionVideo.("height") := 216;

regionCaptions := HASH_TABLE;
regionCaptions.("left") := 130;   regionCaptions.("width") := 220;
regionCaptions.("top") := 229;   regionCaptions.("height") := 20;

regionMenu := HASH_TABLE;
regionMenu.("left") := 2; regionMenu.("width") := 84;
regionMenu.("top") := 7; regionMenu.("height") := 260;

regionunnamed_region := HASH_TABLE; regionunnamed_region.("left") := 0;
regionunnamed_region.("top") := 0;

regionunnamed_region_1 := HASH_TABLE; regionunnamed_region_1.("left") := 0;
regionunnamed_region_1.("top") := 0;

PAR
BEGIN
    SEQ
    BEGIN
       PAR
       BEGIN
         SEQ
         BEGIN
```

```
      imgFBT := new IMAGE( ); imgFBT.region := regionVideo; imgFBT.start := 5;
      imgFBT.duration := "4s"; imgFBT.source := "In-1.gif";

      imgFBT-0 := new IMAGE( ); imgFBT-0.region := "Video"; imgFBT-0.start := 0.9;
      imgFBT-0.duration := "3.2s"; imgFBT-0.source := "In-2.gif";

      imgFBT-1 := new IMAGE( ); imgFBT-1.region := "Video"; imgFBT-1.start := 2;
      imgFBT-1.duration := "15s"; imgFBT-1.source := "In-3.gif";
  END;
  audHmGeb := new AUDIO( ); audHmGeb.region := regionunnamed_region_1;
  audHmGet.source := HappyBirthday.mp3;
  imgSkipIntro := new IMAGE( ); imgSkipIntro.source := Skip.png;
  imgSkipIntro.region := regionMenu.("top") + 193;
END;

PAR
BEGIN
   imgF1s := new IMAGE( );
   imgF1s.region := regionMenu+5; imgF1s.source := "F1s.jpg";
   imgF1s.size := <81,54>;

   imgW1s := new IMAGE( );
   imgW1s.region := regionMenu+66; imgW1s.source := "W1s.jpg";
   imgW1s.size := <80,53>;

   imgA1s    := new IMAGE( );
   imgA1s.region := regionMenu+123; imgA1s.source := "A1s.jpg";
   imgA1s.size := <81,54>;

   imgFBTs   := new IMAGE( );
   imgFBTs.region := regionMenu+195; imgFBTs.source := "FBTs.jpg";
   imgFBTs.size := <81,54>;

   SEQ
BEGIN
   PAR
   BEGIN
      vidFz3_0 := new VIDEO( );
      vidFz3_0.region := regionVideo; vidFz3_0.source := Fz3-g.mpg;

      audBirthday := new AUDIO( );
      audBirthday.region := regionaudio; audBirthday.source := Birthday.mp3;
   END;

IF(imgF1s.Click) THEN
    PAR
    BEGIN
        vidFz3_0 := new VIDEO( );
        vidFz3_0.region := regionVideo; vidFz3_0.source := Fz3-g.mpg;

        audBirthday := new AUDIO( );
        audBirthday.region := regionaudio; audBirthday.source := Birthday.mp3;
    END;
ELSIF(imgW1s.Click) THEN
   PAR
   BEGIN
     vidW1s_1 := new VIDEO( );
     vidW1s_1.region := regionVideo; vidW1s_1.source := Fz3-g.mpg;

     audBirthday_0 := new AUDIO( );
     audBirthday_0.region := regionaudio; audBirthday_0.source := Birthday.mp3;
   END;
ELSIF(imgA1s.Click) THEN
   PAR
```

```
            BEGIN
              vidAz3_0 := new VIDEO( );
              vidAz3_0.region := regionVideo; vidAz3_0.source := Fz3-g.mpg;

              audBirthday_1 := new AUDIO( );
              audBirthday_1.region := regionaudio; audBirthday_1.source := Birthday.mp3;
            END;
          ELSIF(imgFBTs.Click) THEN
            PAR
            BEGIN
              SEQ
              BEGIN
                imgFBT_n := new IMAGE( ); imgFBT_n.region := regionVideo;
                imgFBT_n.dur := "5.9s"; imgFBT_n.source := "FBT.jpg";
                vidBTz3 := new VIDEO( ); vidBTz3.region := regionVideo;
                vidBTz3.source := "BTz3-g.mpg";
                imgFBP := new IMAGE( );   imgFBP.region := regionVideo;
                imgFBP.source := "BPz3-g.mpg";
              END
              audVORBOKFJ := new AUDIO( );
              audVORBOKFJ.region := regionaudio; audVORBOKFJ.src := "Ballgame.mp3";
            END;
          FI
        END;
      END
      imgBkgdImg := new IMAGE( );
      imgBkgdImg.region := regionbkgd_image; imgBkgdImg.src := "Back3s.gif";
    END
END
```

Figure 7-16 Translated WSL Program of the Multimedia Application

In order to perform the multimedia program abstraction related to the spatial feature or the temporal feature, the transformations guided by the expertise and the domain features are used with the evaluation of the abstractness features and the multimedia domain-specific metrics. In this case, the target to abstraction the temporal feature will be experimented as an example. For this target, the following metrics in Table 7-6 are selected and contained in the target model. The three metrics positively contribute the abstractness target.

Metrics	Definition
ABST-STAT	The percentage of statements at higher abstract levels over the total statements
ABST-VOC	The percentage of constructs at higher abstract levels in the total constructs in the program
PTR	The percentage of temporal constructs in the total constructs in the program

Table 7-6 Selected Metrics for the Multimedia Application Abstraction

As the target is quite specific, the expertise on using transformation for abstraction is

essential for this work. By referring the expertise rules given in Chapter 6, the rules can be altered according to the domain features of the multimedia program as follows.

(1) Remove the irrelevant statements to the temporal feature;

(2) Remove ghost variables which have no effect on the execution of the program;

(3) Change the data representation to a more abstract representation;

(4) Replace references to the concrete variables by references to the abstraction variables;

Transformation \ Metric	Cost	Benefit			Impact
		ABST-STAT	ABST-VOC	PTR	
Before Transformation	—	—	—	0.165	—
T95 (P1, posn1) ∧ T95 (P1, posn2) ∧ ... ∧T95 (P1, posnm)	0.989	0	0	0.372	0.652
T15 (P2)	0	0	0	0.422	0
T52 (P3)	0.894	0.279	0.132	0.521	0.947
T97 (P4)	0.707	0.563	0.324	0.712	0.704

T95: Temporal_Abstraction; T15: Delete_All_Redundant; T52: Prog_To_Spec;
T97: Replace_by_Concrete;

Table 7-7 Impact of the Selected Transformations on the Case Study 3

The given rules present the guidance for the transformation prediction. In this circumstance, the prediction process does not adopt the metrics based heuristics but the expertise incorporated and domain specific approach. However, the transformations used for abstraction and other purpose, such as simplifying or rewriting, are also tested if their condition functions satisfice the state function of the program. Therefore, by attempting the applicable transformations, the evaluation data can be generated as follows. The transformations predicted by the expertise rules.

Figure 7-17 is the result after applying the transformation T52 (P3) that transforms the

program to the specification.

```
Begin
Parallel(
    Sequence(
        Parallel(
                Sequence((imgFBT := new IMAGE( ); imgFBT.start := 5; imgFBT.duration := "4s");
                        (imgFBT-0 := new IMAGE( ); imgFBT-0.start := 0.9; imgFBT-0.duration := "3.2s");
                        (imgFBT-1 := new IMAGE( ); imgFBT-1.start := 2; imgFBT-1.duration := "15s"))
                || (audHmGeb := new AUDIO( );)
                || (imgSkipIntro := new IMAGE( ););
        Parallel ((imgF1s := new IMAGE( ))
                || (imgW1s := new IMAGE( ))
                || (imgA1s     := new IMAGE( ))
                || (imgFBTs    := new IMAGE( ))
                || (Sequence(
                        Parallel((vidFz3_0 := new VIDEO( ))||(audBirthday := new AUDIO( ));
                        IF(imgF1s.Click) THEN
                            Parallel((vidFz3_0 := new VIDEO( ))||audBirthday := new AUDIO( ));
                        ELSIF(imgW1s.Click) THEN
                            Parallel((vidW1s_1 := new VIDEO( ))||audBirthday_0 := new AUDIO( ));
                        ELSIF(imgA1s.Click) THEN
                            Parallel((vidAz3_0 := new VIDEO( ))||audBirthday_1 := new AUDIO( ));
                        ELSIF(imgFBTs.Click) THEN
                            Parallel(
                                Sequence((imgFBT_n := new IMAGE( ); imgFBT_n.dur := "5.9s");
                                        (vidBTz3 := new VIDEO( ));
                                        (imgFBP := new IMAGE( )))
                                || (audVORBOKFJ := new AUDIO( ))
                            )
                        )
        )
    )
    || (imgBkgdImg := new IMAGE( ))
)
```

Figure 7-17 Transformation Result from Program to Specification

Through the expertise incorporated algorithm and the domain features, the transformation steps predicted are

T95 (P1, posn1) ∧ T95 (P1, posn2) ∧ ... ∧T95 (P1, posnm) ; T15 (P2) ; T52 (P3) ; T97(P4)

Figure 7-18 gives the final result of the abstraction process. The result is the specification related to the temporal features of the multimedia application. From the specification, it is easy to understand the temporal relations between the media data.

```
(
  (imgFBT > imgFBT_0 >imgFBT_1)  o  (audHmGeb)  o  (imgSkipIntro))
   > (imgF1s  =  imgW1s  =  imgA1s  =  imgFBTs)
       o  ((vidFz3_0  =  audBirthday)
           >
           (imgF1s.Click  s  (vidFz3_0  =  audBirthday) OR
            imgW1s.Click  s  (vidW1s_1  =  audBirthday_0) OR
            imgA1s.Click  s  (vidAz3_0  =  audBirthday_1) OR
            imgFBTs.Click  s  ((imgFBT_n > vidBTz3 > imgFBP)  =  audVORBOKFJ)
            )
           )
          )
   o  imgBkgdImg
)
```

Figure 7-18 Abstraction Result of the Transformations

7.7 Summary

The chapter shows the tool prototype support for the proposed approach and experiments the proposed approach on three case studies, including a procedural program by the metrics based algorithm, an object-oriented program by incorporating expertise and a multimedia application in SMIL by the algorithm involved in the domain features.

- The toolset FermaT Transformation Predictor (F-TP) is integrated into the FermaT Integration Platform (FIP) and composed of four modules, i.e. the extended WSL parser, the target modeller, the metrics viewer and the transformation predictor.

- The toolset has not been mature enough to apply for the practical application but supplied an academic prototype to demonstrate the transformation prediction approach. The efficiency of toolset still needs to improve.

- The case on the procedural program by the metric based algorithm shows how the transformation paths are generated and ranked without the help of the expertise. The case shows the proposed approach is more promising, efficient and correct than the traditional approach without using the prediction means.

- The case on the object-oriented program shows how the transformation paths containing the parallel transformations are predicted.

- The case on the multi-media program shows how the expertise incorporated algorithm and the utilisation of domain features are used in the transformation prediction process.

- The reengineering targets can be achieved by the proposed transformation prediction approach.

- From the three case studies, it is concluded that the expertise incorporated algorithm is more efficient than the metric guided algorithm if the expertise algorithm can be used.

- The validation of the predicted result and the transformed result is performed by software engineer.

Chapter 8
Conclusion and Future Work

Objectives

- To summarise the thesis and give the conclusions
- To evaluate the research described in the thesis
- To illustrate the limitation of the work
- To propose the future work

8.1 Summary of the Thesis

The thesis aims to improve and augment the applications of program transformation for software reengineering. A systematic method for using program transformation in software reengineering, called Target-Driven Program Transformation Step Prediction framework (TDPTSP), is proposed to achieve the aim.

In the framework, in order to realise the aim to improve the implementation of transformation for determined reengineering targets, the target model referring the goal-driven technique [89] is used to specify the targets for which the program transformations are implemented. In the model, the relevant software metrics are

selected to measure the status of the source code and evaluate the impact of the transformations applied. By using this model, the suitable transformation candidates and their execution sequence are predicted through a precisely heuristic based algorithm. The aim to augmenting the program transformation with domain features is experimented by extending the WSL and its transformation theory into the multimedia domain.

The features of the proposed approach (including the toolset) are listed as follows.

- Using WSL as an intermediate language to explore the theory and implementation of program transformation for software reengineering.

- Utilising the hierarchical characteristics of the WSL constructs to extend the language. To distinguish the existing constructs, called the basic level of WSL, the extended levels are named as secondary level.

- Using the fix-point theory to extend WSL with object-oriented features. The extension covers the basic object-oriented concepts, such as class, object, instantiation and reference.

- Extending WSL with multimedia features by the extended object-oriented features. The media are modelled as classes in WSL. The spatial, the temporal and the logical relations are defined as constructs of WSL.

- Proposing the transformations based on object-oriented extension of WSL.

- Proposing the transformations based on multimedia extension of WSL.

- Defining the mathematical notations of program transformation in order to

manage the transformation process.

- Extending the transformation bank which is a library contained in the transformation engine and giving the meta-model of transformation.

- Classifying the transformations according to the effects of their operations.

- Defining the concept of target in the proposed approach.

- Selecting and justifying six categories of reengineering intensive software metrics which are used to measure reengineering target.

- Building the target model which represents the relations of targets, target factors and metrics related to the target.

- Correlating targets, metrics and transformations within a united model (MOTMET) and give the formula to compute the target satisficed degree.

- Formulating the transformation prediction as using heuristic based algorithm to construct a transformation process model which includes the ranked solutions.

- Incorporating expertise into the prediction algorithm.

- Predicting transformations for the domain specific applications by taking the domain features into account.

- Developing the toolset based on the transformation engine and investigating the proposed transformation prediction approach in three case studies on a procedural program, an object-oriented program and a multimedia program.

8.2 Conclusion

To conclude this thesis: a target driven program transformation approach for software reengineering is proposed. The work is based on WSL which is a program transformation-intensive language. The reengineering activities are represented as target models which correlate software metrics. WSL is extended on both syntax and semantics with the new features, including object-oriented and multimedia domain features, so that the proposed work can be applied in the domain specific application. The prediction algorithms including metrics based algorithm, the expertise incorporated prediction algorithm as well as taking domain features into account, are proposed to improve efficiency and correctness of the target driven program transformation process for reengineering. A supporting prototype FermaT Transformation Predictor (F-TP), which implements the approach, based on FermaT Integration Platform (FIP) is developed to speed and scale up practical reengineering.

8.3 Evaluation of Research Questions

In Chapter 1, a set of research questions were proposed as criteria to judge the success of the approach described in this thesis. In this section, the detailed analyses of our approach are presented based on these criteria.

- ✓ *What are the advantages of the implementation of reengineering with the program transformation prediction approach against the one without using the approach?*

The aim of the transformation step prediction based reengineering approach is to improve the efficiency and automation of the transformation process for reengineering target. Traditionally, without such automation assistant means, there might be some obstacles in the process, such as:

- ☒ The program transformation steps must be predefined by an expert who is familiar with the transformation system and understand the program throughout.

- ☒ The lack of quantitative evaluation of the process makes the assessment of the transformation result for the reengineering targets hard.

- ☒ In most case, the needs of reengineering imply a number of criteria not just a single one therefore taking all the relevant reengineering concerns into account is impossible by using the traditional approach even if implemented by an expert.

Equipped with the proposed approach, the advantages against the above obstacles can be summarised as follows.

- ☑ The proposed transformation prediction approach can provide clues for doing the practical reengineering of software. With the approach, the software engineer can find the proper candidates to implement transformations for the given targets. It can be much more efficient than searching the transformations by the experience manually only. The prediction does not mean automating the transformation process completely because the prediction is to search a set of solutions and it is the software engineer to determine which solution is applied. The software engineer can benefit from learning and applying the predicted result.

- ☑ The utilisation of software metrics provides the quantitative measurement. The measurement can be used to assess the impact of a target and guide the transformation process towards the target.

- ☑ Traditional program transformation process without the assistant of prediction normally only follows a predefined step which is determined according to experience or a standard method. With the proposed approach, a set of potential solutions could be obtained automatically.

- ☑ The solutions generated by the approach can be used to research the order of the transformation sequence which can be useful information for the development of new transformations.

- ☑ If the state of a program matches the condition of the expertise rules, a preferential solution can be predicted straightway. This facilitates the automation and efficiency of the transformation process.

✓ *Are the extended constructs consistent with the syntax and semantics of WSL?*

The extension starts from the object-oriented constructs by using the fix-point theory. The denotational semantics of the object-oriented constructions is consistent with the existing semantics of WSL. The specification of the new constructions is also presented. When a multimedia application is reengineered, the media data is translated to objects of WSL. The control of the media data is translated to the extended constructs of WSL. Therefore, the extension is consistent with WSL on the semantics and syntax.

✓ *Is the target model correct and complete to represent the identified target?*

A target could include the sub-targets, which are affected by several factors. They are identified according to the experience and common knowledge of the software engineer. The target model covers the relevant sub-targets and the relations between them. The model has to be validated by human being, rather than by machine.

✓ *Can the prediction be modelled as a search problem?*

To provide the useful information guiding the transformation process, the transformation engine is supposed to determine the suitable candidates and predict the impact of the transformations. The transformation candidates are chosen by an automated process from the transformation bank that contains a large number of transformations. Therefore, the prediction can be performed via a search operation to obtain the answer.

✓ *What kind heuristics will be used in the proposed approach? Is the heuristic knowledge useful for the transformation prediction?*

Heuristic search strategies that use some kind of additional (heuristic) information can reduce the computational costs for many search problem instances. The heuristics used in the proposed prediction algorithm are three kinds of knowledge, i.e. the relations between metrics and program transformations, the expertise obtained in practice and the domain features. Without the heuristic knowledge, the procedure of constructing the transformation process model will have a very heavy burden to expend the model by adding all of the available transformations. The heuristics can facilitate the process by pruning the transformation paths, which could negatively impact the desired target. Especially when a state of a program matches a pattern and conditions of an expertise rule, the transformation process model can become more explicit and straightforward. By investigating the experiments, the efficiency of using the expertise as the heuristics can be improved approximately 25% by comparing the one without using the expertise. Furthermore, the predicted transformation steps generated by the expertise are more reliable than the metrics based approach because it is from the experts who apply the transformations in practice. Therefore, the answer to the second question is positive.

✓ *How can the quantitative approach be used to control the transformation prediction process?*

The usage of software metrics provides a quantitative means to evaluate the impact of transformations. A computation formula is given for this purpose. The formula is constructed by taking into four factors account: (1) the program feature change resulted by a transformation. The difference of metric value caused by this change is referred in the formula; (2) the normalisation of the calculation; (3) the relations of metrics modelled in the target model; (4) the cost of the transformation execution. With the four considerations, the formula is complete to measure the impact of a transformation.

✓ *Will the prediction result be a good solution?*

The answer is positive for the transformations within the finite steps and the traversed levels. The process is guided by the heuristics and quantitative measurement based on a well-formed representation of the desired target. Hence, it is more efficient to approach the required constraints by applying transformations. However, the correctness of the result could be negatively affected if the target model is constructed inappropriately. This should be addressed by introducing a more formal approach for the target representation. It can be considered in the future work.

8.4 Limitations

Although the proposed approach has a number of advantages and facilitates the reengineering process, there are limitations with it. Because of the program transformation's nature that preserves semantics and alters syntax, the target to changing software functions driven by evolution needs cannot be fulfilled by the program transformation prediction approach. To achieve such a target, more techniques

are needed, such as re-specification. In addition, in the proposed approach, there are some technique steps prone to subjectivity. For example, the target modelling relies on the user's knowledge about the desired target. Any misunderstanding of the target will result in an incorrect result. The solution of this problem will be worked on in the future work.

8.5 Future Work

The research presented in this thesis is not the terminus. The following future work can be pursued based on the present work.

- Modelling Target

As a center of the proposed approach, the target modelling is performed based on the discussion about the relations between targets, metrics and transformations. In the proposed approach, the target is modelled by using the goal-driven technique. AND/OR relations are included in the model. In terms of the level of abstraction, a target is a kind of requirement which is at much high abstraction level. The validation of the model depends on the experience of software engineering so that this process is very 'subjective'. In the future work, a more formal and precise technique is needed for this representation. With regard to this consideration, ontology [60] could be used to introduce more precise relations for the target modelling.

- Generating heuristic relations between metrics and transformation

Heuristics are essential for improving the transformation step prediction. The relations between metrics and transformations are used as a heuristic used in the construction of the transformation process model so that only the transformations which positively

impact the target are included. This heuristic is generated according to the definition of transformations and their possible impacts on the program features with regard to selected metrics. However, this is not precise enough to ensure the actual relations between the two entities are captured. Therefore, a statistical approach could be suggested in the future for this purpose. The approach is to simulate the running of a great number of test cases in which how a transformation affects program features and the selected metrics can be reviewed and accordingly obtain more accurate results. Using the result, the transformation prediction can be more promising.

- Obtaining expertise rules

Expertise rules are the ones gained from the practical work and experimented as accepted strategy to follow. In the thesis, several expertise rules have been introduced and represented. However, there are still more existing expertise rules that are not included in the proposed approach. To explore a complete expertise library is also an aim of the future work.

References

[1] G. Agosta, G. Palermo and C. Silvano, "Multi-Objective Co-Exploration of Source Code Transformations and Design Space Architectures for Low-Power Embedded Systems", *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Nicosia, Cyprus, March, 2004, pp.891-896.

[2] A. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques and Tools*: Addison-Wesley, 1986.

[3] R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis and K. R. Luecke, "Reengineering C++ Component Models via Automatic Program Transformation", *Proceedings of the 12th Working Conference on Reverse Engineering* Pittsburgh, PA, USA, November 2005, pp.13-22.

[4] R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis and K. R. Luecke, "Case Study: Re-engineering C++ Component Models via Automatic Program Transformation", *Information & Software Technology*, vol. 49, 2007, pp. 275-291.

[5] J. F. Allen, "Maintaining Knowledge about Temporal Intervals", *Communications of the ACM*, vol. 26, 1983, pp. 832-843.

[6] A. W. Appel, *Modern Compiler Implementation in ML*: Cambridge University Press, 1998.

[7] G. Arango, I. D. Baxter, C. Pidgeon and P. Freeman, "TMM: Software Maintenance by Transformation", *IEEE Software*, vol. 3, 1986, pp. 27-39.

[8] R. Arnold, "A Road Map Guide to Software Reengineering", *Software Reengineering*: IEEE Computer Society Press, 1994.

[9] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring", *Proceedings of Working Conference on Reverse Engineering*, Queensland, Australia, November 2000, pp.98-107.

[10] R. Balzer, "A 15 Year Perspective on Automatic Programming", *IEEE*

Transactions on Software Engineering, vol. 11, 1985, pp. 1257-1267.

[11] I. D. Baxter, "Transformational maintenance by Reuse of Design Histories", *PhD Thesis*: University of California at Irvine, Irvine, USA, 1990.

[12] I. D. Baxter, "Using Transformation Systems for Software Maintenance and Reengineering", *Proceedings of International Conference on Software Engineering*, Toronto, Canada, May 2001, pp.739-740.

[13] K. Bennett and V. Rajlich, "Software Maintenance and Evolution: a Roadmap", *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp.73-87.

[14] S. Black, "The Role of Ripple Effect in Software Evolution", *Software Evolution and Feedback: Theory and Practice*: Wiley Publishers, 2006.

[15] J. W. Brackett, "Software Requirements", *Technical Report SEI-CMU-19-1.2*, Software Engineering Institute, Carnegie Mellon University, 1990.

[16] M. V. D. Brand and E. Visser, "Generation of Formatters for Context-Free Languages", *ACM Transactions on Software Engineering and Methodology*, vol. 5, 1996, pp. 1-41.

[17] M. Bravenboer, A. v. Dam, K. Olmos and E. Visser, "Program Transformation with Scoped Dynamic Rewrite Rules", *Fundamenta Informaticae*, vol. 69, 2005, pp. 1-56.

[18] M. Bravenboer, R. Vermaas, J. Vinju and E. Visser, "Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax", *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*, Tallinn, Estonia, September 2005, pp.157--172.

[19] D. C. A. Bulterman and L. Rutledge, *SMIL2.0 Interactive Multimedia for Web and Mobile Devices*, Berlin Heidelberg New York: Springer, 2004.

[20] R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs", *Journal of the ACM (JACM)*, vol. 24, 1977, pp. 44-67.

[21] F. Chen, M. Ladkau, S. Li and S. Natelberg, "Technical Report: FermaT

Intergrated Platform (FIP) Development Strategy", *Technical Report*, Software Technology Research Laboratory, De Montfort University, Leicester, UK, December 2005.

[22] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph", *Proceedings of International Workshop Program Comprehension (IWPC'00)*, Limerick, Ireland, June 2000, pp.241-249.

[23] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, vol. 20, 1994, pp. 476 - 493.

[24] E. J. Chikofsky and J. H. C. II, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, vol. 7, 1990, pp. 13-17.

[25] L. K. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*: Kluwer Academic Publishers, 2000.

[26] E. Clementini, P. D. Felice and P. v. Oosterom, "A Small Set of Formal Topological Relationships Suitable for End-User Interaction", *Proceedings of the 3rd International Symposium on Advances in Spatial Databases* Singapore, June 1993, pp.277-295.

[27] W. R. Cook and J. Palsberg, "A Denotational Semantics of Inheritance and Its Correctness", *Information and Computation*, vol. 114, 1994, pp. 329-350.

[28] K. D. Cooper, P. J. Schielke and D. Subramanian, "Optimizing for Reduced Code Space Using Genetic Algorithms", *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, Atlanta, Georgia, United States, May 1999, pp.1-9.

[29] J. Cordy, T. Dean, A. Malton and K. Schneider, "Software Engineering by Source Transformation - Experience with TXL", *Proceedings of International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, November 2001, pp.168-178.

[30] Centrum Voor Wiskunde en Informatica (CWI), "The AMBULANT Player", *http://www.cwi.nl/projects/Ambulant/*, 2007.

[31] S. Demeyer, "Maintainability versus Performance: What's the Effect of

Introducing Polymorphism?" Universiteit Antwerpen (UA), Antwerpen, Belgium, September 2002.

[32] J. C. Deprez and A. Lakhotia, "A Formalism to Automate Mapping from Program Features to Code", *Proceedings of International Workshop on Program Comprehension (IWPC)*, Limerick, Ireland, June 2000, pp.72-83.

[33] A. v. Deursen, J. Heering and P. Klint, "Language Prototyping: An Algebraic Specification Approach", *AMAST Series in Computing*, vol. 5, 1995.

[34] E. W. Dijkstra, *A Discipline of Programming*, Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[35] K. Ding, K. Zhou, F. He and Y. Shen:, "LDA - A Java-Based Linkage Disequilibrium Analyzer", *Bioinformatics*, vol. 19, 2003, pp. 2147-2148

[36] C. Djeraba, *Multimedia Mining, a Highway to Intelligent Multimedia Documents*: Kluwer Publicshers, 2003.

[37] S. Ducasse, M. Rieger and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code", *Proceedings of International Conference on Software Maintenance (ICSM'99)*, Oxford, UK, August, 1999, pp.109-118.

[38] T. Eisenbarth, R. Koschke and D. Simon, "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29, 2003, pp. 210-224.

[39] R. Fanta and V. Rajlich, "Reengineering Object-Oriented Code", *Proceedings of International Conference of Software Maintenance*, Bethesda, MD, USA, November 1998, pp.238-246.

[40] R. Fanta and V. Rajlich, "Restructuring Legacy C Code into C++", *Proceedings of International Conference of Software Maintenance*, Oxford, UK, August 1999, pp.77-85.

[41] D. Fatiregun, M. Harman and R. M. Hierons, "Evolving Transformation Sequences using Genetic Algorithms", *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, Chicago, IL, USA, September 2004, pp.66-75.

[42] M. S. Feather, "A Survey and Classification of Some Program Transformation

Approaches and Techniques", *Proceedings of Working Conference on Program Specification and Transformation*, Ailz, Germany, 1987, pp.165-195.

[43] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach (2nd edition)*, London: International Thomson Computer Press, 1998.

[44] M. Fowler, K. Beck, J. Brant and W. Opdyke, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.

[45] P. Giorgini, J. Mylopoulos, E. Nicchiarelli and R. Sebastiani, "Reasoning with Goal Models", *Proceedings of the 21st International Conference on Conceptual Modeling, Lecture Notes in Computer Science 2503*, Tampere, Finland, October 2002.

[46] F. Glover, "Tabu Search: A Tutorial", *Interface*, vol. 20, 1990, pp. 74-94.

[47] M. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study", *Proceedings of International Conference on Software Maintenance (ICSM)*, San Jose, California, USA, October 2000, pp.131-142.

[48] D. E. Goldberg, *Genetic Algorithms in Search, Optimisation and Machine Learning*, Reading, MA: Addison-Wesley, 1989.

[49] J. Guyon, P.-E. Moreau and A. Reilles, "An Integrated Development Environment for Pattern Matching Programming", *Proceedings of 2nd eclipse Technology eXchange workshop*, Barcelona, Spain, April 2004.

[50] M. Halstead, "Natural Laws Controlling Algorithm Structure?" *ACM SIGPLAN Notices*, vol. 7, 1972, pp. 19-26.

[51] M. Harman and J. A. Clark, "Metrics Are Fitness Functions Too", *Proceedings of the10th IEEE International Software Metrics Symposium (METRICS'04)*, Chicago, IL, USA, September 2004, pp.58-69.

[52] M. Harman and B. Jones, "Search Based Software Engineering", *Journal of Information and Software Technology*, vol. 43, 2001, pp. 833-839.

[53] M. Harman and J. Wegener, "Getting Results from Search-Based Approaches to Software Engineering", *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, UK, May 2004, pp.728-729.

[54] J. L. Hein, *Discrete Mathematics (2nd Edition)*, London: Jones and Bartlett Publishers International, 2003.

[55] IEEE, "IEEE Standard for a Software Quality Metrics Methodology", 1998.

[56] ISO, "ISO 9241-11: Guidance on Usability ", 1998.

[57] ISO/IEC, "ISO/IEC JTC1, ISO-9126-1: Software Engineering - Product Quality - Part 1: Quality Model", 2005.

[58] JavaCC, "Java Compiler Compiler", in *http://javacc.dev.java.net/*.

[59] Jia Liu, D. S. Batory and C. Lengauer:, "Feature oriented refactoring of legacy applications", *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[60] H. Kaiya and M. Saeki, "Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach", *Proceedings of the 5th IEEE International Conference on Quality Software (QSIC)*, Melboume, Australia, September 2005, pp.223-230.

[61] Y. Kataoka, M. D. Ernst, W. G. Griswold and D. Notkin, "Automated Support for Program Refactoring Using Invariants", *Proceedings of IEEE International Conference of Software Maintenance (ICSM)*, Florence, Italy, November 2001, pp.736-743.

[62] R. Kazman and S. Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence", *Technical Report*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 1997.

[63] S. E. Keller, J. A. Perkins, T. F. Payton and S. P. Mardinly, "Tree Transformation Techniques and Experiences", *Proceedings of ACM SIGPLAN Symposium on Compiler Construction*, Montreal, Canada, June 1984, pp.190-201.

[64] B. L. Kovits, *Practical Software Requirements: A Manual of Content and Style*, Greenwitch, CT: Manning Publications Co., 1999.

[65] D. H. Krantz and R. D. Luce, *Foundations of Measurement*: Academic Press, 1971.

[66] H. E. Kyburg, *Theory and measurement*, Cambridge: Cambridge University Press, 1984.

[67] P. J. M. v. Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Practice*, Dordrecht, the Netherlands: Kluwer Academic Publishers, 1987.

[68] M. M. Lehman and J. F. Ramil, "Software Evolution -- Background, Theory, Practice", *Information Processing Letters*, vol. 88, 2003, pp. 33-44.

[69] X. Li and G. Zheng, "A Modified Inheritance Mechanism Enhancing Reusability and Maintainability in Object-Oriented Languages", *Proceedings of the 3rd IEEE Asia-Pacific Software Engineering Conference(APSEC)*, Seoul, South Korea, December, pp.93-102.

[70] G. Lindén, H. Tirri and A. I. Verkamo, "ALCHEMIST: a General Purpose Transformation Generator ", *Source Software—Practice & Experience Archive*, vol. 26, 1995, pp. 653-675.

[71] L. Liu and E. Yu, "Designing Information Systems in Social Context: a Goal and Scenario Modelling Approach", *Information Systems*, vol. 29, 2004, pp. 187-203.

[72] X. Liu, "Abstraction: A Notion for Reverse Engineering", in *Software Technology Research Laboratory*, *PhD Thesis*: De Montfort University, Leicester, UK, 1999.

[73] A. C. Luther, *Authoring Interactive Multimedia*: Morgan-Kaufman Pub, 1994.

[74] T. J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol. 2, 1976, pp. 308-320.

[75] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*: Microsoftware Press, 1993.

[76] T. Mens and T. Tourwe, "A Survey of Software Refactoring", *IEEE Transactions on Software Engineering*, vol. 30, 2004.

[77] A. Metha and G. T. Heineman, "Evolving Legacy System Features into Fine-Grained Components", *Proceedings of International Conference of Software Engineering (ICSE2002)*, Orlando, Florida, USA, May 2002.

[78] R. Millham, "Evolution of Batch-Oriented COBOL Systems into Object-Oriented Systems through Unified Modelling Language", in *Software Technology Research Laboratory, PhD Thesis*: De Montfort University, Leicester, UK, 2005.

[79] D. Millicev, "Domain Mapping Using Extended UML Object Diagrams", *IEEE Software*, vol. 19, 2002, pp. 90-97.

[80] P.-E. Moreau, C. Ringeissen, M. Vittek and I. G. Hedin, "A Pattern Matching Compiler for Multiple Target Languages", *Proceedings of 12th Conference on Compiler Construction*, Warsaw, Poland, May 2003, pp.61-76.

[81] R. E. Mortimer and K. H. Bennett, "Maintenance and Abstraction of Program Data Using Formal Transformations", *Proceedings of International Conference on Software Maintenance*, Monterey, CA, USA, November 1996, pp.301.

[82] S. Muchnick, *Advanced Compiler Design and Implementation*: Morgan-Kaufman Publishers, 1997.

[83] J. Mylopoulos, L. Chung and B. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", *IEEE Transactions on Software Engineering*, vol. 18, 1992, pp. 483-497.

[84] J. Mylopoulos, L. Chung and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis", *Communications of the ACM*, vol. 42, 1999, pp. 31-37.

[85] M. R. Olsem, "An Incremental Approach to Software System Re-engineering", *Journal of Software Maintenance: Research and Practice*, vol. 10, 1998, pp. 181-202.

[86] W. F. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks", *PhD Thesis*: University of Illinois, Urbana-Champaign, USA, 1992.

[87] Oratrix, "GRiNS for SML", *www.oratrix.com*, 2007.

[88] C. R. Pandian, *Software Metrics: A Guide to Planning, Analysis, and Application*: Auerbach Publications, 2004.

[89] R. E. Park, W. B. Goethert and W. A. Florac, "Goal-Driven Software Measurement? A Guidebook", *Technical Report*, Software Engineering Institute, Carnegie Mellon University, 1996.

[90] I. Pashov, "Feature-Based Methodology for Supporting Architecture Refactoring and Maintenance of Long-Life Software Systems", *PhD thesis*: Technical University of Ilmenau, Ilmenau, Germany, 2004.

[91] A. Pettorossi and M. Proietti, "Program Transformation: Theoretical Foundations and Basic Techniques (Part1)", *Fundamenta Informaticae*, vol. 66, 2005, pp. i-iii.

[92] R. S. Pressman, *Software Engineering — A Practitioner's Approach*, New York: McGraw-Hill, 2005.

[93] Program-Transformation.Org, "Program Transformation Wiki", 2002.

[94] M. Proietti and A. Pettorossi, "Semantics Preserving Transformation Rules for Prolog", *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, New Haven, Connecticut, USA, September 1991, pp.274-284.

[95] F. S. Roberts, *Measurement Theory with Applications to Decision Making, Utility, and the Social Sciences*, Reading, MA: Addison-Wesley, 1979.

[96] L. H. Rosenberg and L. E. Hyatt, "Software Re-engineering", *Technical Report SATC-TR-95-1001*, NASA Software Assurance Technology Center, Washington, USA, October 1996.

[97] C. Ryan, *Automatic Re-engineering of Software Using Genetic Programming*, vol. 2: Kluwer Academic Publishers, 1999.

[98] D. Sands, "Total Correctness by Local Improvement in the Transformation of Functional Programs", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, 1996, pp. 175-234.

[99] K. Slonneger and B. L. Kurtz, *Formal Syntax and Semantics of Programming Languages*: Addison-Wesley Publishing Company, 1995.

[100] D. R. Smith, "KIDS: A Semiautomatic Program Development System", *IEEE*

Transactions on Software Engineering, vol. 16, 1990, pp. 1024-1043.

[101] R. Steinmetz and K. Nahrstedt, *Multimedia Applications*, Berlin, Germany: Springer-Verlag, 2004.

[102] B. S. Stewart and C. C. White, "Multiobjective A*", *Journal of the ACM(JACM)*, vol. 38, 1991, pp. 775-814.

[103] L. Tahvildari, "Quality-Driven Object-Oriented Reengineering Framework", *PhD Thesis*: University of Waterloo, Waterloo, Ontario, Canada, 2003.

[104] L. Tahvildari and K. Kontogiannis, "A Software Transformation Framework for Quality-Driven Object-Oriented Re-engineering", *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, Montreal, Canada, October 2004, pp.596-605.

[105] S. Tilley, "Perspectives on Legacy Systems Reengineering", Reengineering Center, Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, USA, 1995.

[106] E. Visser, "A Survey of Rewriting Strategies in Program Transformation Systems", *Proceedings of Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, Utrecht, the Nethelands, May 2001.

[107] E. Visser, Z.-e.-A. Benaissa and A. Tolmach, "Building Program Optimizers with Rewriting Strategies", *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, Baltimore, Maryland, USA, September 1998, pp.13--26.

[108] E. Visser, T. Mens and M. Wallace, "Program Transformation Wiki", in *http://www.program-transformation.org/Transform/ProgramTransformation*, 2004.

[109] W3C, "XML Path Language (XPath) Version 1.0. W3C Recommendation ", *Proceedings of http://www.w3.org/TR/xpath*, November 1999.

[110] W3C, "XSL Transformation (XSLT) Version 1.0. W3C Recommendation ", *Proceedings of http://www.w3.org/TR/xslt*, November 1999.

[111] M. Ward, "Proving Program Reginements and Transformations", *PhD Thesis*:

Oxford University, Oxford, UK, 1989.

[112] M. Ward, "A Definition of Abstraction", *Journal of Software Maintenance: Research and Practice*, vol. 7, 1995, pp. 443-450.

[113] M. Ward, "Program Analysis by Formal Transformation ", *The Computer Journal*, vol. 39, 1996, pp. 598-618.

[114] M. Ward, "Assembler to C Migration Using the FermaT Transformation System", *Proceedings of International Conference of Software Maintenance (ICSM)* Oxford, UK, August 1999, pp.67-76.

[115] M. Ward, "The FermaT Assembler Re-engineering Workbench", *Proceedings of International Conference of Software Maintenance (ICSM)*, Florence, Italy, November 2001, pp.659-662.

[116] M. Ward, "WSL Manual ", *FermaT Transformation Engine3*, 2002.

[117] M. Ward, "Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations", *Science of Computer Programming, Special Issue on Program Transformation*, vol. 52/1-3, 2004, pp. 213-255.

[118] M. Ward and H. Zedan, "MetaWSL and Meta-Transformations in the FermaT Transformation System", *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, Edinburgh, UK, July 2005.

[119] M. Ward and H. Zedan, "Slicing as a Program Transformation", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, 2006.

[120] B. Wegbreit, "Goal-Directed Program Transformation", *IEEE Transactions on Software Engineering*, vol. 2, 1976, pp. 69-80.

[121] J. V. Wijngaarden, "Code Generation from a Domain Specific Language", vol. Master Thesis: Utrecht University, 2003.

[122] N. Wilde, M. Buckellew, H. Page, V. Rajlich and L. Pounds, "A Comparison of Methods for Locating Features in Legacy Software", *Journal of Systems and Software*, 2002, pp. 105-114.

[123] W. E. Wong, S. S. Gokhale, J. R. Horgan and K. S. Trivedi, "Locating Program Features using Execution Slices", *Proceedings of the 1999 IEEE Symposium on Application-specific Systems and Software Engineering Technology*, Richardson, Texas, USA, March 1999, pp.194-203.

[124] H. Yang, P. Luker and W. C. Chu, "Measuring Abstractness for Reverse Engineering in a Re-engineering Tool", *Proceedings of the International Conference on Software Maintenance (ICSM)*, Bari, Italy, September 1997, pp.48-57.

[125] H. Yang and M. Ward, *Successful Evolution of Software Systems*, Boston, London: Arteth House Computing Library, 2003.

[126] X. Yao, "Details of Grant of EPSRC Project SEBASE: Software Engineering By Automated Search", *http://gow.epsrc.ac.uk/ViewGrant.aspx?GrantRef=EP/D052785/1*, 2006.

[127] T. I. Yoshio Kataoka, Hiroki Andou, Tetsuji Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring", *Proceedings of International Conference of Software Maintenance (ICSM'02)*, Montreal, Quebec, Canada, October 2002.

[128] E. Younger, Z. Luo, K. H. Bennett and T. M. Bull, "Reverse Engineering Concurrent Programs Using Formal Modelling and Analysis", *Proceedings of International Conference of Software Maintenance (ICSM)*, Monterey, CA, USA, November 1996, pp.255-264.

[129] E. Younger and M. Ward, "Understanding Concurrent Program using Program Transformations", *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, July 1993, pp.160-168.

[130] Y. Yu, J. C. Leite, J. Mylopoulos, L. L. Liu, E. Yu and E. D. Hollander, "Software Refactoring Guided by Multiple Soft-Goals", *Proceedings of the First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE), in conjunction with the 10th Working Conference on Reverse Engineering*, Victoria, Canada, November 2003.

[131] W. Zhao, L. Zhang, Y. Liu, J. Sun and F. Yang, "SNIAFL: Towards a Static Non-Interactive Approach to Feature Location", *Proceedings of International Conference on Software Engineering (ICSE)*, Edinburgh, UK, May 2004.

[132] S. Zhou and H. Yang, "Measuring Software Components through Object-Orientation and Abstraction for Reengineering", *Proceedings of the ACM and IEEE International Symposium on Internet Technology (ISIT), Taibei, 1998.*

[133] Y. Zou, "Techniques and Methodologies for the Migration of Legacy Systems to Object Oriented Platforms", *PhD Thesis*: University of Waterloo, Ontario, Canada, 2003.

[134] Y. Zou and K. Kontogiannis, "Incremental Transformation of Procedural Systems to Object Oriented Platforms", *Proceedings of the 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC)*, Dallas, Texas, USA, November 2003, pp.290-295.

Appendix A
Backus Naur Form of Extended WSL

```
WSL        ::=  statements <EOF>
statements ::=  statement ( <S_SEMICOLON> statement )*
statement  ::=  stat_if
            |   stat_d_if
            |   stat_d_do
            |   stat_while
            |   stat_do
            |   stat_exit
            |   stat_for
            |   stat_var
            |   stat_comment
            |   stat_assert
            |   stat_assignment
            |   stat_push
            |   stat_pop
            |   stat_join
            |   stat_actions
            |   stat_call
            |   stat_print
            |   stat_mw_func_decl
            |   stat_begin
            |   stat_foreach
            |   stat_ateach
            |   stat_ifmatch
            |   stat_ifmatch2
            |   stat_maphash
            |   stat_error
            |   stat_spec
            |   stat_single_assign
            |   stat_pattern
            |   stat_proc_call
            |   stat_require
            |   stat_class
            |   stat_seq
            |   stat_par
            |   ( <S_SKIP> )
            |   ( <S_ABORT> )
            |   ( <S_STAT_PLACE> )
stat_if    ::=  ( ( <S_IF> condition <S_THEN> statements ) ( ( <S_ELSIF> condition <S_THEN>
            statements ) )* ( ( <S_ELSE> statements <S_FI> ) | pseudo_else <S_FI> ) )
stat_d_if  ::=  ( ( <S_D_IF> condition <S_ARROW> statements ) ( ( <S_BOX> condition <S_ARROW>
            statements ) )* <S_FI> )
stat_d_do  ::=  ( ( <S_D_DO> condition <S_ARROW> statements ) ( ( <S_BOX> condition <S_ARROW>
            statements ) )* <S_OD> )

stat_while ::=  ( <S_WHILE> condition <S_DO> statements <S_OD> )
stat_do    ::=  ( <S_DO> statements <S_OD> )
stat_exit  ::=  T_Exit
stat_for   ::=  ( <S_FOR> T_Var_Lvalue <S_BECOMES> expression <S_TO> expression <S_STEP>
            expression <S_DO> statements <S_OD> )
            |   ( <S_FOR> T_Var_Lvalue <S_IN> s_expression <S_DO> statements <S_OD> )
stat_var   ::=  ( <S_VAR> <S_LANGLE> assigns <S_RANGLE> <S_COLON> statements <S_ENDVAR> )
```

```
stat_comment    ::=   T_Comment
stat_assert ::=   ( <S_LBRACE> condition <S_RBRACE> )
stat_assignment ::=   ( <S_LANGLE> assigns <S_RANGLE> )
stat_push   ::=   ( <S_PUSH> <S_LPAREN> T_Var_Lvalue <S_COMMA> s_expression <S_RPAREN> )
stat_pop    ::=   ( <S_POP> <S_LPAREN> T_Var_Lvalue <S_COMMA> T_Var_Lvalue <S_RPAREN> )
stat_join   ::=   ( <S_JOIN> statements <S_COMMA> statements <S_ENDJOIN> )

stat_actions::=   ( <S_ACTIONS> T_IdentifierName <S_COLON> actions <S_ENDACTIONS> )
stat_call   ::=   T_Call

stat_print  ::=   ( <S_PRINT> <S_LPAREN> expressions <S_RPAREN> )
            |     ( <S_PRINFLUSH> <S_LPAREN> expressions <S_RPAREN> )
stat_mw_func_decl   ::=   ( <S_MW_PROC> T_AtName <S_LPAREN> ( ( lvalue ( <S_COMMA> lvalue )* )* )
            var_lvalues <S_RPAREN> <S_DEFINE> statements ( <S_END> | <S_FULLSTOP> ) )
            |     ( <S_MW_FUNCT> T_AtName <S_LPAREN> ( ( lvalue ( <S_COMMA> lvalue )* )* )
            <S_RPAREN> <S_DEFINE> ( ( <S_VAR> <S_LANGLE> assigns <S_RANGLE> <S_COLON>
            statements <S_SEMICOLON> <S_LPAREN> expression <S_RPAREN> ( <S_END> |
            <S_FULLSTOP> ) ) | ( <S_COLON> statements <S_SEMICOLON> <S_LPAREN> expression
            <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) ) )
            |     ( <S_MW_BFUNCT> T_AtName <S_QUERY> <S_LPAREN> ( ( lvalue ( <S_COMMA>
            lvalue )* )* ) <S_RPAREN> <S_DEFINE> ( ( <S_VAR> <S_LANGLE> assigns <S_RANGLE>
            <S_COLON> statements <S_SEMICOLON> <S_LPAREN> condition <S_RPAREN> ( <S_END> |
            <S_FULLSTOP> ) ) | ( <S_COLON> statements <S_SEMICOLON> <S_LPAREN> condition
            <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) ) )
stat_begin  ::=   ( <S_BEGIN> statements <S_WHERE> defines <S_END> )
stat_foreach::=   <S_FOREACH> ( ( <S_STATEMENT> <S_DO> statements <S_OD> ) | ( <S_STATEMENTS>
            <S_DO> statements <S_OD> ) | ( <S_VARIABLE> <S_DO> statements <S_OD> ) | ( <S_GLOBAL>
            <S_VARIABLE> <S_DO> statements <S_OD> ) | ( <S_LVALUE> <S_DO> statements <S_OD> ) |
            ( <S_STS> <S_DO> statements <S_OD> ) | ( <S_NAS> <S_DO> statements <S_OD> ) |
            ( <S_EXPRESSION> <S_DO> statements <S_OD> ) | ( <S_CONDITION> <S_DO> statements
            <S_OD> ) | ( <S_TERMINAL> ( ( <S_STATEMENT> <S_DO> statements <S_OD> ) |
            ( <S_STATEMENTS> <S_DO> statements <S_OD> ) ) ) )
stat_ateach ::=   <S_ATEACH> ( ( <S_STATEMENT> <S_DO> statements <S_OD> ) | ( <S_STATEMENTS>
            <S_DO> statements <S_OD> ) | ( <S_VARIABLE> <S_DO> statements <S_OD> ) | ( <S_GLOBAL>
            <S_VARIABLE> <S_DO> statements <S_OD> ) | ( <S_LVALUE> <S_DO> statements <S_OD> ) |
            ( <S_STS> <S_DO> statements <S_OD> ) | ( <S_NAS> <S_DO> statements <S_OD> ) |
            ( <S_EXPRESSION> <S_DO> statements <S_OD> ) | ( <S_CONDITION> <S_DO> statements
            <S_OD> ) | ( <S_TERMINAL> ( ( <S_STATEMENT> <S_DO> statements <S_OD> ) |
            ( <S_STATEMENTS> <S_DO> statements <S_OD> ) ) ) )
stat_ifmatch      ::=   <S_IFMATCH> ( ( <S_STATEMENTS> statements <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_STATEMENT>
            statement <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements
            <S_ENDMATCH> ) ) ) | ( <S_EXPRESSION> expression <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_EXPRESSIONS>
            expressions <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements
            <S_ENDMATCH> ) ) ) | ( <S_CONDITION> condition <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_DEFINITION>
            define <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements
            <S_ENDMATCH> ) ) ) | ( <S_DEFINITIONS> defines <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_ASSIGN> assign
            <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) |
            ( <S_ASSIGNS> assigns_node <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE>
            statements <S_ENDMATCH> ) ) ) | ( <S_ACTION> action <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_GUARDED>
            guarded <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements
            <S_ENDMATCH> ) ) ) | ( <S_LVALUE> lvalue <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) |
            ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_LVALUES> lvalues <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) )
stat_ifmatch2     ::=   <S_IFMATCH2> ( ( <S_STATEMENTS> statements <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_STATEMENT>
            statement <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements
            <S_ENDMATCH> ) ) ) | ( <S_EXPRESSION> expression <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_EXPRESSIONS>
            expressions <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements
```

```
            <S_ENDMATCH> ) ) ) | ( <S_CONDITION> condition <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_DEFINITION>
            define <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements
            <S_ENDMATCH> ) ) ) | ( <S_DEFINITIONS> defines <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_ASSIGN> assign
            <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) |
            ( <S_ASSIGNS> assigns_node <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE>
            statements <S_ENDMATCH> ) ) ) | ( <S_ACTION> action <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_GUARDED>
            guarded <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements
            <S_ENDMATCH> ) ) ) | ( <S_LVALUE> lvalue <S_THEN> statements ( ( ( <S_ENDMATCH> ) ) |
            ( <S_ELSE> statements <S_ENDMATCH> ) ) ) | ( <S_LVALUES> lvalues <S_THEN> statements
            ( ( ( <S_ENDMATCH> ) ) | ( <S_ELSE> statements <S_ENDMATCH> ) ) ) )
stat_maphash    ::=   ( <S_MAPHASH> <S_LPAREN> T_Name <S_COMMA> expression <S_RPAREN> )
stat_error  ::=   ( <S_ERROR> <S_LPAREN> expressions <S_RPAREN> )
stat_spec   ::=   ( <S_SPEC> <S_LANGLE> lvalues <S_RANGLE> <S_COLON> condition <S_ENDSPEC> )
stat_proc_call    ::=   ( T_IdentifierName <S_LPAREN> ( ( expression ( <S_COMMA> expression )* )* )
            var_lvalues <S_RPAREN> )
            |     ( <S_PLINK_P> T_IdentifierName <S_LPAREN> ( ( expression ( <S_COMMA>
            expression )* )* ) var_lvalues <S_RPAREN> )
            |     ( T_AtName ( <S_LPAREN> )* ( ( expression ( <S_COMMA> expression )* )* ) var_lvalues
            ( <S_RPAREN> )* )
            |     ( T_AtPatOneName <S_LPAREN> ( ( expression ( <S_COMMA> expression )* )* ) var_lvalues
            <S_RPAREN> )
            |     ( <S_PLINK_XP> T_IdentifierName <S_LPAREN> ( ( expression ( <S_COMMA>
            expression )* )* ) <S_RPAREN> )
stat_pattern      ::=   T_Stat_Pat_One
            |     T_Stat_Pat_Many
            |     T_Stat_Pat_Any
stat_single_assign      ::=   ( ( lvalue <S_BECOMES> expression ) )
            |     ( lvalue <S_FULLSTOP> <S_LPAREN> expression <S_RPAREN> <S_BECOMES>
            expression )
guarded   ::=   ( condition ( <S_THEN> | <S_ARROW> ) statements )
            |     ( ( T_Cond_Pat_One | T_Cond_Pat_Many | T_Cond_Pat_Any ) ( <S_THEN> | <S_ARROW> )
            statements )
defines   ::=   define ( ( <S_COMMA> define ) | ( define ) )*
define    ::=   ( stat_func_decl | T_Defn_Pat_One | T_Defn_Pat_Many | T_Defn_Pat_Any )
stat_func_decl    ::=   ( <S_PROC> T_IdentifierName <S_LPAREN> ( ( lvalue ( <S_COMMA> lvalue )* )* )
            var_lvalues <S_RPAREN> <S_DEFINE> statements ( <S_END> | <S_FULLSTOP> ) )
            |     ( <S_FUNCT> T_IdentifierName <S_LPAREN> ( ( lvalue ( <S_COMMA> lvalue )* )* )
            <S_RPAREN> <S_DEFINE> ( ( <S_VAR> <S_LANGLE> assigns <S_RANGLE> <S_COLON>
            <S_LPAREN> expression <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) | ( <S_COLON>
            <S_LPAREN> expression <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) ) )
            |     ( <S_BFUNCT> T_IdentifierName <S_QUERY> <S_LPAREN> ( ( lvalue ( <S_COMMA>
            lvalue )* )* ) <S_RPAREN> <S_DEFINE> ( ( <S_VAR> <S_LANGLE> assigns <S_RANGLE>
            <S_COLON> <S_LPAREN> condition <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) |
            ( <S_COLON> <S_LPAREN> condition <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) ) )
stat_require::=   ( <S_REQUIRE> ((lvalues ( <S_COMMA> lvalues)*)*)
stat_class  ::=   ( <S_CLASS> T_IdentifierName (<S_COLON> T_IdentifierName ) <S_LBRACE>
            ( ( field_value ( <S_COMMA> field_value)* ) (method <S_COMMA> method)*) ( <S_RBRACE>
            (<S_END> | <S_FULLSTOP>) ) ) )
stat_seq    ::=   ( <S_SEQ> <S_BEGIN> (lvalue ( <S_COMMA> lvalue )* )* )

stat_par    ::=   ( <S_PAR> <S_BEGIN> (lvalue ( <S_D_LINE> lvalue )* )* )
field_value::=    (<S_FIELD> ( lvalue ( <S_COMMA> lvalue )* )* )* )
method    ::=   ( <S_METHOD> T_IdentifierName <S_LPAREN> ( ( lvalue ( <S_COMMA> lvalue )* )* )
            var_lvalues <S_RPAREN> <S_DEFINE> statements ( <S_END> | <S_FULLSTOP> ) )
actions   ::=   ( <S_METHOD> T_IdentifierName <S_LPAREN> ( ( lvalue ( <S_COMMA> lvalue )* )* )
            <S_RPAREN> <S_DEFINE> ( ( <S_VAR> <S_LANGLE> assigns <S_RANGLE> <S_COLON>
            <S_LPAREN> expression <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) | ( <S_COLON>
            <S_LPAREN> expression <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) ) )
action    ::=   ( <S_METHOD> T_IdentifierName <S_QUERY> <S_LPAREN> ( ( lvalue ( <S_COMMA>
            lvalue )* )* ) <S_RPAREN> <S_DEFINE> ( ( <S_VAR> <S_LANGLE> assigns <S_RANGLE>
            <S_COLON> <S_LPAREN> condition <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) |
```

```
            ( <S_COLON> <S_LPAREN> condition <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) ) )
assigns_node::= ( assign ( <S_COMMA> assign )* )
assigns     ::=  assign ( <S_COMMA> assign )*
assign      ::=  ( T_Var_Lvalue <S_BECOMES> expression )
var_lvalues::=  ( ( <S_VAR> ( lvalue ( <S_COMMA> lvalue )* )* )* )
lvalues     ::=  ( lvalue ( <S_COMMA> lvalue )* )
lvalue      ::=  ( T_Var_Lvalue | T_Lvalue_Pat_One | T_Lvalue_Pat_Many | T_Lvalue_Pat_Any )
            ( <S_LBRACKET> a_expressions <S_RBRACKET> | <S_LBRACKET> a_expression <S_DOTDOT>
            ( <S_RBRACKET> | a_expression <S_RBRACKET> ) | <S_LBRACKET> a_expression
            <S_COMMA> a_expression <S_RBRACKET> | ( T_Struct_Lvalue ) )*
condition   ::=  ( b_term ( <S_OR> b_term )* )
b_term      ::=  ( b_factor ( <S_AND> b_factor )* )
b_factor    ::=  ( <S_NOT> b_factor )
            |    b_atom
b_atom      ::=  ( <S_LPAREN> condition <S_RPAREN> )
            |    ( <S_TRUE> )
            |    ( <S_FALSE> )
            |    ( <S_COND_PLACE> )
            |    rel_exp   |   cond_pat   |   cond_prefix
cond_pat    ::=  T_Cond_Pat_One
            |    T_Cond_Pat_Many
            |    T_Cond_Pat_Any
rel_exp     ::=  expression ( <S_EQUAL> expression | <S_NEQ> expression | <S_LANGLE> expression |
            <S_RANGLE> expression | <S_LEQ> expression | <S_GEQ> expression | <S_IN> expression |
            <S_NOTIN> expression )
cond_prefix     ::=  ( <S_EVEN> <S_QUERY> <S_LPAREN> expression <S_RPAREN> )
            |    ( <S_ODD> <S_QUERY> <S_LPAREN> expression <S_RPAREN> )
            |    ( <S_SUBSET> <S_QUERY> <S_LPAREN> s_expression <S_COMMA> s_expression
            <S_RPAREN> )
            |    ( <S_MEMBER> <S_QUERY> <S_LPAREN> expression <S_COMMA> s_expression
            <S_RPAREN> )
            |    ( T_IdentifierName <S_QUERY> <S_LPAREN> ( ( expression ( <S_COMMA> expression )* )* )
            <S_RPAREN> )
            |    ( T_AtName <S_QUERY> <S_LPAREN> ( ( expression ( <S_COMMA> expression )* )* )
            <S_RPAREN> )
            |    ( <S_PLINK_XC> T_IdentifierName <S_QUERY> <S_LPAREN> ( ( expression
            ( <S_COMMA> expression )* )* ) <S_RPAREN> )
expressions::=  ( expression ( <S_COMMA> expression )* )
expression  ::=  a_expression
            |    fill_expression
            |    fill2_expression
            |    if_expression
            |    new_expression
            |    multi_expression
if_expression   ::=  ( <S_IF> condition <S_THEN> expression <S_ELSE> expression <S_FI> )
fill_expression ::=  ( <S_FILL> <S_STATEMENTS> statements <S_ENDFILL> )
            |    ( <S_FILL> <S_STATEMENT> statement <S_ENDFILL> )
            |    ( <S_FILL> <S_EXPRESSION> expression <S_ENDFILL> )
            |    ( <S_FILL> <S_EXPRESSIONS> expressions <S_ENDFILL> )
            |    ( <S_FILL> <S_CONDITION> condition <S_ENDFILL> )
            |    ( <S_FILL> <S_DEFINITION> define <S_ENDFILL> )
            |    ( <S_FILL> <S_DEFINITIONS> defines <S_ENDFILL> )
            |    ( <S_FILL> <S_ASSIGN> assign <S_ENDFILL> )
            |    ( <S_FILL> <S_ASSIGNS> assigns_node <S_ENDFILL> )
            |    ( <S_FILL> <S_ACTION> action <S_ENDFILL> )
            |    ( <S_FILL> <S_GUARDED> guarded <S_ENDFILL> )
            |    ( <S_FILL> <S_LVALUE> lvalue <S_ENDFILL> )
            |    ( <S_FILL> <S_LVALUES> lvalues <S_ENDFILL> )
fill2_expression ::=  ( <S_FILL2> <S_STATEMENTS> statements <S_ENDFILL> )
            |    ( <S_FILL2> <S_STATEMENT> statement <S_ENDFILL> )
            |    ( <S_FILL2> <S_EXPRESSION> expression <S_ENDFILL> )
            |    ( <S_FILL2> <S_EXPRESSIONS> expressions <S_ENDFILL> )
            |    ( <S_FILL2> <S_CONDITION> condition <S_ENDFILL> )
            |    ( <S_FILL2> <S_DEFINITION> define <S_ENDFILL> )
```

```
            |    ( <S_FILL2> <S_DEFINITIONS> defines <S_ENDFILL> )
            |    ( <S_FILL2> <S_ASSIGN> assign <S_ENDFILL> )
            |    ( <S_FILL2> <S_ASSIGNS> assigns_node <S_ENDFILL> )
            |    ( <S_FILL2> <S_ACTION> action <S_ENDFILL> )
            |    ( <S_FILL2> <S_GUARDED> guarded <S_ENDFILL> )
            |    ( <S_FILL2> <S_LVALUE> lvalue <S_ENDFILL> )
            |    ( <S_FILL2> <S_LVALUES> lvalues <S_ENDFILL> )
new_expression ::=  ( <S_NEW> T_IdentifierName <S_LPAREN> lvalues <S_RPAREN>)
multi_expression::= ( lvalue <T_Rel> lvalue )
s_expression    ::=  a_expression
a_expressions   ::=  a_expression
a_expression    ::=  term ( <S_PLUS> term | <S_MINUS> term | <S_CONCAT> term | <S_UNION> term )*
term        ::=  factor ( <S_TIMES> factor | <S_SLASH> factor | <S_MOD> factor | <S_DIV> factor |
            <S_BACKSLASH> factor )*
factor      ::=  ( true_factor | ( <S_MINUS> factor ) ) ( <S_CARET> a_expression | <S_CARET> <S_CARET>
            s_expression | <S_LBRACKET> a_expressions <S_RBRACKET> | <S_LBRACKET> a_expression
            <S_DOTDOT> ( <S_RBRACKET> | a_expression <S_RBRACKET> ) | <S_LBRACKET>
            a_expression <S_COMMA> a_expression <S_RBRACKET> | ( T_Struct ) | <S_FULLSTOP>
            <S_LPAREN> expression <S_RPAREN> )*
true_factor ::=  exp_atom ( <S_EXPONENT> factor )*
exp_atom    ::=  <S_LPAREN> a_expression <S_RPAREN>
            |    T_Number
            |    a_prefix_op
            |    ( T_IdentifierName <S_LPAREN> ( ( expression ( <S_COMMA> expression )* )* )
            <S_RPAREN> )
            |    ( T_AtName <S_LPAREN> ( ( expression ( <S_COMMA> expression )* )* ) <S_RPAREN> )
            |    ( <S_PLINK_XF> T_IdentifierName <S_LPAREN> ( ( expression ( <S_COMMA>
            expression )* )* ) <S_RPAREN> )
            |    <S_EXPN_PLACE>
            |    <S_VAR_PLACE>
            |    T_Expn_Pat_One
            |    T_Expn_Pat_Many
            |    T_Expn_Pat_Any
            |    T_Variable
            |    T_String
            |    T_Set
            |    T_Sequence
            |    numb_type
            |    s_prefix_op
a_prefix_op::=  ( <S_ABS> <S_LPAREN> a_expression <S_RPAREN> )
            |    ( <S_FRAC> <S_LPAREN> a_expression <S_RPAREN> )
            |    ( <S_INT> <S_LPAREN> a_expression <S_RPAREN> )
            |    ( <S_SGN> <S_LPAREN> a_expression <S_RPAREN> )
            |    ( <S_MAX> <S_LPAREN> a_expression <S_COMMA> a_expression <S_RPAREN> )
            |    ( <S_MIN> <S_LPAREN> a_expression <S_COMMA> a_expression <S_RPAREN> )
            |    ( <S_LENGTH> <S_LPAREN> s_expression <S_RPAREN> )
            |    ( <S_REDUCE> <S_LPAREN> T_Name <S_COMMA> s_expression <S_RPAREN> )
            |    ( <S_HEAD> <S_LPAREN> s_expression <S_RPAREN> )
            |    ( <S_LAST> <S_LPAREN> s_expression <S_RPAREN> )
T_Set       ::=  ( <S_LBRACE> expression <S_VBAR> condition <S_RBRACE> )
T_Sequence::=  ( <S_LANGLE> ( ( expression ( <S_COMMA> expression )* )* ) <S_RANGLE> )
T_Rel       ::=  <T_LogicRel>
            |    <T_TemporalRel>
            |    <T_SpatialRelation>
T_LogicRel      ::=  <S_SUB>
            |    <S_SUPER>
T_TemporalRel ::=  <S_BEFORE>
            |    <S_AFTER>
            |    <S_MEET>
            |    <S_MET_BY>
            |    <S_T_EQUAL>
            |    <S_T_OVERLAP>
            |    <S_T_OVERLAPPED_BY>
            |    <S_DURING>
```

```
            |   <S_CONTAINS>
            |   <S_STARTS>
            |   <S_STARTED_BY>
            |   <S_FINISHES>
            |   <S_FINISHED_BY>
T_SpatialRel    ::=   <S_DISJOINT>
            |   <S_JOINT>
            |   <S_S_OVERLAP>
            |   <S_COVERS>
            |   <S_INSIDE>
            |   <S_EQUAL>
numb_type ::=   <S_RATS>
            |   <S_REALS>
            |   <S_NATS>
            |   <S_INTS>
s_prefix_op     ::=   ( <S_MAP> <S_LPAREN> T_Name <S_COMMA> s_expression <S_RPAREN> )
            |   ( <S_POWERSET> <S_LPAREN> s_expression <S_RPAREN> )
            |   ( <S_TAIL> <S_LPAREN> s_expression <S_RPAREN> )
            |   ( <S_BUTLAST> <S_LPAREN> s_expression <S_RPAREN> )
            |   ( <S_SLENGTH> <S_LPAREN> s_expression <S_RPAREN> )
            |   ( <S_SUBSTR> <S_LPAREN> expressions <S_RPAREN> )
            |   ( <S_INDEX> <S_LPAREN> expressions <S_RPAREN> )
            |   ( <S_REDUCE> <S_LPAREN> T_Name <S_COMMA> s_expression <S_RPAREN> )
            |   ( <S_HEAD> <S_LPAREN> s_expression <S_RPAREN> )
            |   ( <S_LAST> <S_LPAREN> s_expression <S_RPAREN> )
T_Cond_Pat_One      ::=   <S_PAT_ONE> <S_IDENTIFIER>
T_Cond_Pat_Many     ::=   <S_PAT_MANY> <S_IDENTIFIER>
T_Cond_Pat_Any      ::=   <S_PAT_ANY> <S_IDENTIFIER>
T_Expn_Pat_One      ::=   <S_PAT_ONE> <S_IDENTIFIER>
T_Expn_Pat_Many     ::=   <S_PAT_MANY> <S_IDENTIFIER>
T_Expn_Pat_Any      ::=   <S_PAT_ANY> <S_IDENTIFIER>
T_Lvalue_Pat_One    ::=   <S_PAT_ONE> <S_IDENTIFIER>
T_Lvalue_Pat_Many   ::=   <S_PAT_MANY> <S_IDENTIFIER>
T_Lvalue_Pat_Any    ::=   <S_PAT_ANY> <S_IDENTIFIER>
T_Stat_Pat_One      ::=   <S_PAT_ONE> <S_IDENTIFIER>
T_Stat_Pat_Many     ::=   <S_PAT_MANY> <S_IDENTIFIER>
T_Stat_Pat_Any      ::=   <S_PAT_ANY> <S_IDENTIFIER>
T_Action_Pat_One    ::=   <S_PAT_ONE> <S_IDENTIFIER>
T_Action_Pat_Many   ::=   <S_PAT_MANY> <S_IDENTIFIER>
T_Action_Pat_Any    ::=   <S_PAT_ANY> <S_IDENTIFIER>
T_Defn_Pat_One      ::=   <S_PAT_ONE> <S_IDENTIFIER>
T_Defn_Pat_Many     ::=   <S_PAT_MANY> <S_IDENTIFIER>
T_Defn_Pat_Any      ::=   <S_PAT_ANY> <S_IDENTIFIER>
T_String            ::=   <S_STRING>
T_Number            ::=   <S_NUMBER>
T_Variable          ::=   <S_IDENTIFIER>
T_Name              ::=   <S_STRING>
T_IdentifierName    ::=   <S_IDENTIFIER>
T_AtName            ::=   <S_AT> ( <S_IDENTIFIER> | <S_AMBIGOUS_IDENTIFIER> )
T_AtPatOneName      ::=   <S_AT_PAT_ONE> ( <S_IDENTIFIER> | <S_AMBIGOUS_IDENTIFIER> )
T_Var_Lvalue        ::=   <S_IDENTIFIER>
T_Exit      ::=   <S_EXIT> <S_LPAREN> <S_NUMBER> <S_RPAREN>
T_Comment       ::=   <S_COMMENT> <S_STRING>
T_Call      ::=   <S_CALL> <S_IDENTIFIER>
T_Struct_Lvalue ::=   <S_FULLSTOP> T_IdentifierName
T_Struct    ::=   <S_FULLSTOP> T_IdentifierName
```

Appendix B
XML-based Representation of Target Model

In F-TP, the constructed target model is represented as a diagram and stored in an XML file. Below is there an example of the target model in XML file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<TargetRepository name="Target Repository" version="1.0" idCounter="51"
                  guid="ID_1123C8C13BC000000B3">
  <ModelSpace name="Model" guid="ID_1123C8C13BC000000B4">
    <MetaLevel name="META" guid="ID_1123C8C13BC000000B5">
      <MetaTarget guid="ID_1123C8C13BC000000B6">
      </MetaTarget>
      <MetaTargetNode guid="ID_1123C8C13FA000000D3"/>
      <MetaCardinality guid="ID_1123C8C141A000000E1">
      </MetaCardinality>
      <MetaGroup guid="ID_1123C8C1439000000F0">
      </MetaGroup>
    </MetaLevel>
    <ModelLevel name="MODEL" guid="ID_1123C8C1448000000FF">
<Target name="Component Size" description="" premature="false"
        guid="ID_1123C923DDC00000107" constraintId="7">
</Target>
<SubTarget name="Complexity" description="" premature="false"
          guid="ID_1123CA3AE8500000109" constraintId="9">
</SubTarget>
<SubTarget name="Program Complexity" description="" premature="false"
          guid="ID_1123CA4416E0000010B" constraintId="11">
</SubTarget>
<SubTarget name="Program Nesting Level" description="" premature="false"
           guid="ID_1123CA4804C0000010D" constraintId="13">
</SubTarget>
<Metric name="NCNB" description="" premature="false" impact="negative"
        guid="ID_1123CA4B5950000010F" constraintId="15">
</Metric>
<SubTarget name="Information Flow" description="" premature="false"
          guid="ID_1123CA5475500000115" constraintId="21">
</SubTarget>
<Metric name="RNC" description="" premature="false" impact="negative"
        guid="ID_1123CA5A8EE00000117" constraintId="23">
</Metric>
<Metric name="CFDF" description="" premature="false" impact="negative"
        guid="ID_1123CA62C950000011B" constraintId="27">
</Metric>
<SubTarget name="Internal Control of Structure" description=""
          premature="false" guid="ID_1123CA888C50000012B" constraintId="36">
</SubTarget>
<Metric name="NON" description="" premature="false" impact="negative"
        guid="ID_1123CA8D0F90000012D" constraintId="38">
```

```
</Metric>
<Metric name="McCabe" description="" premature="false" impact="negative"
        guid="ID_1123CA918810000012F" constraintId="40">
</Metric>
<Diagram name="TargetDiagram" guid="ID_1123C8DBDC200000100" xpos="0"
          ypos="0" width="754" height="687">
    <TargetNode guid="ID_1123CA3AE850000010A" refFeatureName="Complexity"
                refGuidFeatureTree="ID_1123CA3AE8500000109" xpos="202"
                ypos="36" constraintId="10">
        <Group guid="ID_1123CA9A91900000137" constraintId="48">
            <Limit min="1" max="1"/>
            <TargetNode guid="ID_1123C923DEB00000108"
                        refFeatureName="Component Size"
                        refGuidFeatureTree="ID_C0A8004100 0001123C923DDC00000107"
                        xpos="55" ypos="96" constraintId="8">
                <Cardinality guid="ID_1123CA6B65600000120" constraintId="32">
                    <Limit min="1" max="1"/>
                    <MetricNode guid="ID_1123CA4B59500000110"
                                refFeatureName="NCNB"
                                refGuidFeatureTree="ID_1123CA4B5950000010F"
                                xpos="1" ypos="241" constraintId="16">
                    </MetricNode>
                </Cardinality>
                <Cardinality guid="ID_1123CA93CA300000131" constraintId="42">
                    <Limit min="1" max="1"/>
                    <MetricNode guid="ID_1123CA8D0F90000012E"
                                refFeatureName="NON"
                                refGuidFeatureTree="ID_1123CA8D0F90000012D"
                                xpos="98" ypos="240" constraintId="39">
                    </MetricNode>
                </Cardinality>
            </TargetNode>
            <TargetNode guid="ID_1123CA4416E0000010C"
                        refFeatureName="Program Complexity"
                        refGuidFeatureTree="ID_1123CA4416E0000010B"
                        xpos="230" ypos="96" constraintId="12">
<Group guid="ID_1123CA9C54C00000138" constraintId="49">
    <Limit min="1" max="1"/>
    <TargetNode guid="ID_1123CA888C50000012C"
                refFeatureName="Internal Control of Structure"
                refGuidFeatureTree="ID_1123CA888C50000012B"
                xpos="176" ypos="162" constraintId="37">
        <Cardinality guid="ID_1123CA94A6F00000132" constraintId="43">
            <Limit min="1" max="1"/>
            <MetricNode guid="ID_1123CA9189100000130"
                        refFeatureName="McCabe"
                        refGuidFeatureTree="ID_1123CA918810000012F"
                        xpos="173" ypos="241" constraintId="41">
            </MetricNode >
        </Cardinality>
    </TargetNode>
    <TargetNode guid="ID_1123CA5475500000116"
                refFeatureName="Information Flow"
                refGuidFeatureTree="ID_1123CA5475500000115"
                xpos="369" ypos="161" constraintId="22">
        <Cardinality guid="ID_1123CA9695100000135" constraintId="46">
            <Limit min="1" max="1"/>
            <MetricNode guid="ID_1123CA62C950000011C"
                        refFeatureName="CFDF"
                        refGuidFeatureTree="ID_1123CA62C950000011B"
                        xpos="470" ypos="236" constraintId="28">
            </MetricNode>
        </Cardinality>
    </TargetNode>
```

```
                    </Group>
                </TargetNode>
                <TargetNode guid="ID_1123CA4804C0000010E"
                            refFeatureName="Program Nesting Level"
                            refGuidFeatureTree="ID_1123CA4804C0000010D"
                            xpos="412" ypos="96" constraintId="14">
                    <Cardinality guid="ID_1123CA9750900000136" constraintId="47">
                        <Limit min="1" max="1"/>
                        <MetricNode guid="ID_1123CA5A8EE00000118"
                                    refFeatureName="RNC"
                                    refGuidFeatureTree="ID_1123CA5A8EE00000117"
                                    xpos="598" ypos="235" constraintId="24">
                        </MetricNode >
                    </Cardinality>
                </TargetNode>
            </Group>
        </TargetNode>
    </Diagram>
  </ModelLevel>
 </ModelSpace>
</TargetRepository>
```

Appendix C
List of Transformations

T0	Abort_Processing	The transformation simplifies statement sequences containing an ABORT.	Simplify
T1	Absorb_Left	The transformation will absorb into the selected statement the one that precedes it.	Join
T2	Absorb_Right	The transformation will absorb into the selected statement the one that follows it.	Join
T3	Actions_To_Where	The transformation converts an action system to a WHERE clause and uses the variable exit_flag to indicate whether the Z action has been called.	Rewrite
T4	Add_Assertion	This transformation will add an assertion after the current item, if some suitable information can be ascertained.	Insert
T5	Add_Left	This transformation will add the selected statement (or sequence of statements) into the statement that precedes it without doing further simplification.	Join
T6	Align_Nested_Statements	This transformation takes a guarded clause whose first statement is a IF and integrates it with the outer condition by absorbing the other guarded statements into the inner IF and then modifying its conditions appropriately. This is the converse of Part.	Rewrite
T7	Apply_to_Right	This transformation will apply the current program item to the one to its immediate right. For example, if the current item is an assertion and the next item is an `IF' statement, then the transformation will attempt to simplify the conditions using the assertions.	Simplify

T8	Collapse_Action_System	The transformation will use simplifications and substitution to transform an action system into a sequence of statements, possibly inside a DO loop.	Rewrite
T9	Collapse_All_Action_Systems	The transformation will attempt to collapse the action systems within a program which is a WHERE structure.	Rewrite
T10	Combine_Wheres	The transformation combines Where Structures' will combine two nested WHERE structures into one structure which will contain the definitions from each of the original WHERE structures.The selected WHERE structure will be merged into an enclosing one if there is one or, failing that, into an enclosed WHERE structure.	Rewrite
T11	Constant_Propagation	Constant Propagation finds assignments of constants to variables in the selected item and propagates the values through the selected item (replacing variables in expressions by the appropriate values)	Simplify
T12	D_Do_To_Floop	The transformation converts a D_DO loop to an equivalent DO..OD loop.	Rewrite
T13	Delete_All_Assertions	This transformation will delete all the `ASSERT' statements within the selected code. If the resulting code is not syntactically correct, the program will be `tidied up' which may well result in the re-instatement of `ASSERT' or `SKIP' statements.	Simplify
T14	Delete_All_Comments	This transformation will delete all the `COMMENT' statements within the selected code. If the resulting code is not syntactically correct, the program will be `tidied up' which may well result in the insertion of `SKIP' statements.	Simplify
T15	Delete_All_Redundant	The transformation searches for redundant statements and deletes all the ones it finds. A statement is `Redundant' if it calls nothing external and the variables it modifies will all be assigned again before their values are accessed.	Delete

T16	Delete_All_Skips	This transformation will delete all the `SKIP' statements within the selected code. If the resulting code is not syntactically correct, the program will be `tidied up' which may well result in the reinstatement of `SKIP' statements.	Simplify
T17	Delete_Item	This transformation will delete a program item that is redundant or unreachable	Delete
T18	Delete_Redundant_Registers	Delete Redundant Registers uses dataflow analysis to find and delete redundant register assignments (assignments to registers which are overwritten or never accessed).	Delete
T19	Delete_Redundant_Statement	The transformation checks whether the current statement is `Redundant' (because it calls nothing external and the variables it modifies will all be assigned again before their values are accessed). If so, it deletes the Statement.	Delete
T20	Delete_Unreachable_Code	The transformation will remove unreachable statements in the selected object. It will also remove unreachable cases in an IF statement, e.g those which follow a TRUE guard	Simplify
T21	Delete_What_Follows	The transformation will delete the code which follows the selected item if it can never be executed	Simplify
T22	Double_to_Single_Loop	The transformation will convert a double nested loop to a single loop, if this can be done without significantly increasing the size of the program.	Rewrite
T23	Else_If_to_Elsif	The transformation will replace an `Else' clause which contains an `If' statement with an `Elsif' clause. The transformation can be selected with either the outer `If' statement, or the `Else' clause selected.	Rewrite
T24	Elsif_To_Else_If	The transformation will replace an `Elsif' clause in an `If' statement with an `Else' clause which itself contains an `If' statement. The transformation can be selected with either the `If' statement, or the `Elsif' clause selected.	Rewrite

T25	Expand_and_Separate	The transformation will expand the selected IF statement to include all the following statements, then separate all possible statements from the resulting IF. This is probably only useful if the IF includes a CALL, EXIT etc. which is duplicated in the following statements, otherwise it will probably achieve nothing.	Rewrite
T26	Expand_and_Separate_All	The transformation will attempt to apply the transformation Expand_and_Separate to the first statement in each action in an action system.	Simplify
T27	Expand_Call	The transformation will replace a call to an action, procedure or function with the corresponding definition.	Rewrite
T28	Expand_Forward	The transformation will copy the following statement into the end of each branch of the selected IF or D_IF statement. It differs from Absorb Right in that the statement is only absorbed into the `top level' of the selected IF.	Join
T29	Find_Terminals	Find and mark the terminal statements in the selected statement. If a terminal statement is a local proc call, apply recursively to the proc body.	Rewrite
T30	Fix_Dispatch	This transformation will search for simple procedures in Assembler code and convert them to WSL PROCs.A simple procedure is a collection of actions with a single entry point and possibly multiple exit points. All the exits are calls to dispatch (ie normal returns), or calls to an action which must lead to an ABEND (ie error returns).	Rewrite
T31	Floop_To_While	The transformation converts a suitable DO...OD loop to a While loop	Rewrite
T32	For_To_While	Convert any FOR loop to a VAR plus WHILE loop	Rewrite
T33	Force_Double_to_Single_Loop	The transformation will convert a double nested loop to a single loop, regardless of any increase in program size which this causes	Rewrite

T34	Fully_Absorb_Right	This transformation will absorb into the selected statement all the statements that follow it.	Join
T35	Fully_Expand_Forward	The transformation applies Expand_Forward as often as possible	Join
T36	Globals_to_Pars	The transformation converts global variables in procs to extra VAR parameters.	Rewrite
T37	Insert_Assertion	This transformation will add an assertion inside the current item, if some suitable information can be ascertained.	Insert
T38	Join_All_Cases	This transformation will join any guards in an `If' statement which contain the same sequence of statements (thus reducing their number) by changing the conditions of all the guards as appropriate.	Rewrite
T39	Make_Proc	The transformation will make a procedure from the body of an action or from a list of statements.	Rewrite
T40	Merge_Call_in_Action	The transformation will attempt to merge calls which call the same action, in the selected action	Simplify
T41	Merge_Calls	The transformation reduces the number of calls in an action system.	Simplify
T42	Merge_Cond_Right	The transformation merges a binary cond with a subsequent Cond which uses the same (or the opposite) test	Simplify
T43	Merge_Left	This transformation will merge the selected statement (or sequence of statements) into the statement that precedes it.	Join
T44	Merge_Right	This transformation will merge the selected statement into the statement that precedes it.	Join
T45	Meta_Trans	Convert a FOREACH with a long sequence of IFMATCH commands to a more efficient form	Simplify

T46	Move_Comment_Left	The transformation moves the selected Comment Left.	Move
T47	Move_Comment_Right	The transformation moves the selected Comment Right.	Move
T48	Move_Comments	The transformation will move any comments which appear at the end of actions within an action system and which follow a call. The comments will be moved in front of the call. This will help tidy up the output of the Herma translator.	Rewrite
T49	Move_to_Left	This transformation will move the selected item to the left so that it is exchanged with the item that precedes it.	Move
T50	Move_to_Right	This transformation will move the selected item to the right so that it is exchanged with the item that follows it.	Move
T51	Partially_Join_Cases	This transformation will join any guards in an IF statement which contain almost the same sequence of statements (thus reducing their number) by introducing a nested IF and changing the conditions of all the guards as appropriate.	Join
T52	Prog_to_Spec	The transformation converts a given program to an equivalent specification statement.	Abstraction
T53	Prune_Dispatch	Simplify the dispatch action by removing references to dest values which do not appear in the rest of the program.	Simplify
T54	Push_Pop	Look for a statement sequence with a PUSH of a var followed by a POP of the same var. Put the sequence inside a VAR to show that the variable is unchanged.	Rewrite
T55	Remove_Recursion_in_Action	Remove Redundant Vars takes out as many local variables as possible from the selected VAR structure. If they can all be taken out, the VAR is replaced by its (possibly modified) body.	Delete

T56	Reduce_Loop	The transformation automatically makes the body of a DO...OD reducible (by introducing new procedures as necessary) and either remove the loop (if it is a dummy loop) or convert the loop to a WHILE loop (if the loop is a proper sequence).	Simplify
T57	Reduce_Multiple_Loops	This transformation will reduce the number of multiply nested loops to a minimum.	Simplify
T58	Refine_Spec	The transformation refines a specification statement into something closer to an implementation	Refinement
T59	Remove_All_Redundant	The transformation applys Remove_Redundant_Vars to every VAR structure in the statement or sequence	Delete
T60	Remove_Galileo_Comments	Removes Galileo comments without a sequence number (SSL or SSE).	Delete
T61	Remove_Dummy_Loop	The transformation will remove a DO loop which is redundant	Simplify
T62	Remove_Redundant_Vars	The transformation takes out as many local variables as possible from the selected VAR structure. If they can all be taken out, the VAR is replaced by its (possibly modified) body.	Delete
T63	Rename_Defns	The transformation renames PROC definitions to avoid name clashes. This allows us to move all the definitions to a single outer WHERE clause.	Rewrite
T64	Rename_Local_Vars	The transformation removes all local VAR statements by renaming the variables.	Rewrite
T65	Rename_Proc	The transformation renames a PROC to given new name	Rewrite

T66	Replace_Accs_with_Value	This transformation will apply Replace_With_Value to all variables with the names a0, a1, a2 and a3 in the selected item.	Rewrite
T67	Replace_with_Value	This transformation will replace a variable (in an expression) by its value -- provided that that value can be uniquely determined at that point in the program.	Rewrite
T68	Restore_Local_Vars	The transformation restores the local var clauses that were converted to global variables by Rename_Local_Vars	Rewrite
T69	Reverse_Order	This transformation will reverse the order of most two-component items; in particular expressions, conditions and **If** which have two branches.	Rewrite
T70	Semantic_Slice	The transformation performs semantic slicing on a subset of WSL. Enter the list of variables to slice on as the data parameter.	Simplify
T71	Separate_Both	The transformation will take code out to the right and the left of the selected structure.	Rewrite
T72	Separate_Left	The transformation will take code out to the left of the selected structure. As much code as possible will be taken out; if all the statements are taken out then the original containing structure will be removed	Rewrite
T73	Separate_Right	The transformation will take code out to the right of the selected structure.	Rewrite
T74	Simple_Action_System	The transformation attempts to remove actions and calls from an action system by successively applying simplifying transformations. As many of the actions as possible will be eliminated without making the program significantly larger.	Simplify

T75	Simplify	This transformation will simplify any component as fully as possible.	Simplify
T76	Simplify_If	The transformation will remove false cases from an IF statement and any cases whose conditions imply earlier conditions. Any repeated statements which can be taken outside the if will be and the conditions will be simplified if possible.	Simplify
T77	Simplify_Item	This transformation will simplify an item, but not recursively simplify the components inside it. In particular, the transformation will simplify expressions, conditions and degenerate conditional, local variable and loop statements.	Simplify
T78	Static_Single_Assignment	The transformation converts WSL code to Static Single Assignment (SSA) form by renaming variables and adding phi function assignments.	Rewrite
T79	Substitute_and_Delete	The transformation will replace all calls to an action, procedure or function with the corresponding definition and delete the definition	Rewrite
T80	Substitute_and_Delete_List	The transformation will replace all calls to any action within the selected list of actions with the corresponding definition and delete the definition. Actions which are called more than once will not be affected.	Rewrite
T81	Syntactic_Slice	The transformation performs Syntactic Slicing using SSA and control dependencies. Enter the list of variables to slice on as the data parameter.	Simplify
T82	Take_out_Left	This transformation will take the selected item out of the enclosing structure towards the left.	Move
T83	Take_Out_of_Loop	This transformation will take the selected item out of an appropriate enclosing loop towards the right.	Move

T84	Take_Out_Right	This transformation will take the selected item out of the enclosing structure towards the right.	Move
T85	Unfold_Proc_Call	The transformation unfolds the selected procedure call, replacing it with a copy of the procedure body.	Rewrite
T86	Unfold_Proc_Calls	The transformation unfolds Proc Calls searches for procedures which are only called once, unfolds the call and removes the procedure.	Simplify
T87	Use_Assertion	If the current item is an assertion, the transformation tries to use the assertion to simplify the following program.	Simplify
T88	Var_Pars_to_Val_Pars	The transformation adds all VAR pars as extra value pars where needed. This is needed by the SSA transformation so that the input and output parameters can get different names.	Rewrite
T89	While_to_Abort	This transformation replaces a non-terminating while loop with a conditional abort.	Simplify
T90	While_to_Floop	The transformation changes a WHILE loop to an equivalent DO..OD loop.	Rewrite
T91	Movement_Transformation	The transformation moves parts of an existing class to a component class and to set up a delegating relationship from the existing class to its component.	Move
T92	Encapsulation_Ttransformation	The transformation is applied when one class creates instances of another and it is required to weaken the association between the two classes by packaging the object creation statements into dedicated methods.	Join
T93	Wrapper_Transformation	The transformation wraps an existing receiver class with another class, in such a way that all requests to an object of the wrapper class are passed to the receiver object it wraps and similarly any results of such requests are passed back by the wrapper.	Join

T94	Spatial_Abstraction	The transformation is to remove the statements which irrelevant to spatial properties.	Delete
T95	Temporal_Abstraction	The transformation is to remove the statements which irrelevant to temporal properties.	Delete
T96	Inverse Transformation)	The transformation is to transform the statement to its inverse representation.	Rewrite
T97	Replace_by_Concrete	The transformation is used to replace the temporal properties by the concrete temporal relations. If the properties of two multimedia objects satisfy one of the following conditions, then the two objects can be transformed to having the corresponding relations.	Abstraction
T98	Transitive_Absorb	If $\boldsymbol{R}$ is transitive, $\exists x, y, z$ satisfies $x\boldsymbol{R}\ y$ *and* $y\boldsymbol{R}\ z$, when only media x and z are concerned, the media y can be absorbed by replaced as $x\boldsymbol{R}\ z$.	Delete
T99	Elimination_Absorb	For $x \in \boldsymbol{A}$, $x\mathbf{Seq}x$ can be transformed as x and $x\mathbf{Par}x$ can be transformed as x.	Delete
T100	Make_Class	The transformation will make a class	Rewrite

Appendix D
List of Publications

F. Chen, **S. Li** and H. Yang, "Enforcing Role-Based Access Controls with Service Oriented Agentification", *the 2007 IEEE International Conference on Networking, Sensing and Control* (*ICNSC2007*), London, UK, April 2007.

F. Chen, **S. Li** and H. Yang, "Feature Analysis for Service-Oriented Reengineering", *the 12th IEEE Asia-Pacific Software Engineering Conference* (*APSEC*), Taibei, December 2005.

S. Li, F. Chen, Z. Liang and H. Yang, "Using Feature-Oriented Analysis to Recover Legacy Software Design for Software Evolution", *the 2005 International Conference of Software Engineering and Knowledge Engineering* (*SEKE*), Taibei, July 2005.

S. Li and H. Yang et al, "Building a Dependable Enterprise Service Assembly Line (ESAL) for Legacy Component Integration", *the 2004 IEEE International Conference on Cyberworlds* (*CW2004*) , Tokyo, Japan, November 2004.

S. Li and H. Yang, "Leveraging Legacy Assets with Enterprise Application Integration Using a Grey-box Modernisation Approach", *the 2004 EPSRC Postgraduate Research Conference in Electronics, Photonics, Communications & Networks and Computing Science* (*PREP*), Hatfield, UK, April 2004.

Z. Liang, **S. Li** and H. Yang et al, "A Multiple-Tier Model Manipulation Architecture on Enterprise Decision Making", *the 27th IEEE Annual International Computer Software*

and Applications Conference (*COMPSAC*), Dallas, Texas, USA, October 2003.

J. Pu, **S. Li** and H. Yang, "Modelling Legacy Code with UML Class", *the 9th Chinese Automation and Computing Society Conference in the UK*, Luton, UK, September 2003.

H. Liao, **S. Li** and H. Yang et al, "Building Dynamical Enterprise Application Expansion Model by Integrated Development Platform", *the 9th Chinese Automation and Computing Society Conference in the UK*, Luton, UK, September 2003.

S. Li, B. Qiao and H. Yang et al, "System Quality Propagation in Reverse Architecturing", *the 7th World Conference on Integrated Design and Process Technology (IDPT),* Austin, Texas, USA, December 2003.

H. Zhou, **S. Li** and H. Yang et al, "A Multiple-tier Distributed Data Integration Architecture", *the 7th World Conference on Integrated Design and Process Technology (IDPT)*, Austin, Texas, USA, December 2003.