Constraint Based Transformation Theory

Abstract

The migration of a legacy system is one of the most challenging tasks in the area of software evolution. There are existing several different approaches to migrate a legacy system with different advantages and disadvantages. One solution is to translate the software into an intermediate language which spans from the low abstraction level to the very high abstraction level. Once the software has been translated to intermediate language the abstraction level can be changed via mathematical proven transformation rules. Afterwards, a translator can easily translate the intermediate language into the target language.

A very powerful intermediate language was invented by Martin Ward at the University of Oxford. This Language is called "Wide Spectrum Language" or "WSL". He also defines a set of mathematical proven code transformations and an environment to apply these transformations to WSL. This environment is called "FermaT Transformation Engine".

In this report we will present initial work on a framework for a constraint based transformation system which gives the maintainer the possibility to apply constraint based code transformations with the FermaT Transformation Engine. It will review some of the achievements and outline the motivation and objectives for the proposed development framework. Furthermore, it reviews literature from the code transformation, constraint programming and expression language domain.

Document Type	Transfer Report
Author	Stefan Natelberg
Date	29th November 2007
Version	1.0
Inspected	Prof. Hussein Zedan

Contents

Chapter 1

Introduction

At the beginning of software development in the early 1950s and 1960s, software was not very complex. The maintenance of software was just a small part of the software life cycle. With some revolutionary hardware developments in the late 1960s and early 1970s software has become more complex and extensive. Maintenance growth to a major part of the software life cycle. At that time, people began to understand that software is not just dying. The re-engineering of software as a part of software maintenance came up for the first time. At the end of the 1970s, the industry was faced with many major problems. The costs of software maintenance began to explode. In some areas the costs of maintenance was taking more effort than the initial development. Thereby, the problem of maintenance was a general problem caused by some properties and the complexity of software systems. The demand for significant changes of software went up. The implementation of the changes themselves created additional problems.

Nowadays, there are existing lots of systems which have to be maintained, extended or redeveloped. These systems, which often have no or inconsistent documentation and are written in a legacy language, are called "legacy systems". Thereby, a legacy system in general is very mature and the rare numbers of bugs are well known. The maintenance of such a system is very expensive because of the complexity and the low-level programming language. On the other hand, it is very dangerous and expensive to replace the complete system by a new one, especially for business applications. Therefore, a migration of the legacy system could be the answer which is one of the most challenging tasks in the area of software evolution.

The migration can be done by humans who analyse an existing program and rewrite it in another language or for other requirements. Possibly, this will produce

the best result but the time consumption would be huge and new errors could arise. It is also possible to translate software with a brute force algorithm which simply picks one source language statement after another and transforms it into one or more statements of the target language. This is a very quick procedure but probably the result will not be very good. The migrated code can be very complex and the lines of code could increase dramatically. This way of software translation can possibly be done if the source language and the target language are quite similar but not if the abstraction levels of these two languages are too far apart. Another solution to migrate software is to translate the software into an intermediate language which spans from the low abstraction level to the very high abstraction level. Once the software has been translated to intermediate language the abstraction level can be changed via mathematical proven transformation rules. Afterwards, a translator can easily translate the intermediate language into the target language.

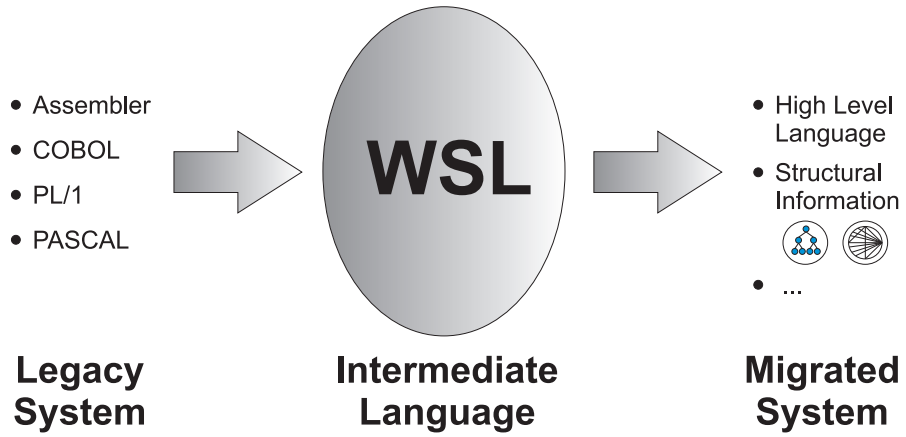

Figure 1.1: WSL as intermediate language for legacy system analysing and abstraction.

In our case, we use the approach of an intermediate language and transformation rules to increase the level of abstraction. The intermediate language is necessary because most languages just support a small spectrum of abstraction. A second advantage of this concept is that it works with a huge number of programming languages. All we need to do to use other languages is to write a translator which is not that difficult. If the migration to the target language has been successful, a compiler can in return generate assembler code.

A very powerful intermediate language was invented by Martin Ward at the

University of Oxford. This Language is called "Wide Spectrum Language" or "WSL". He also defines a set of mathematical proven code transformations and an environment to apply these transformations to WSL. This environment is called "FermaT Transformation Engine". One of the weak points of this work is that the user needs to measure the code after every transformation. Therewith, the process of transformation is not automated and it is still a lot of work for the maintainer to measure the code and apply the transformations to find a transformation sequence which meets his requirements. In the following chapter we will discuss why it is a problem to apply FermaT Transformations in the current system.

Chapter 2

Problem Statement

The process of code transformation is a very difficult part of software migration [21]. One of the main problems is to find out what the particular requirements on the system are. For example, if there is a customer who just wants to migrate code from one machine to another and than do not touch it anymore, it is not important what the code looks like [19]. Unfortunately, this is not the case in general. Customers usually want to migrate the code to a high-level language and than maintain it or add specific functions to the system. Therewith, the code needs to be as simple and structured as possible. In other words, customers want code which is easy to maintain [3]. But what does this mean?

In this case, the main problem is the measuring of code. It is easy to measure code by "number of statements", "McCabe index" or just the "lines of code" [4]. Also, it is not that difficult to count branches, jumps or loops. But how can we measure maintainability? How can we define "simple code"? Probably every customer who wants to migrate one of his legacy systems would define maintainability in another way. Maintainability contains different criteria and can not be defined that easy [10]. But maybe it is possible to split maintainability up in different criteria which can be defined and weighted for each customer in a different way [6].

The measuring of migrated code is one of the problems which occurs during the re-engineering of a legacy system [7, 11, 13]. Another huge problem is the hardware dependence of low level assembler code. Basically, this means that low level assembler code can contain pointers and system calls which cannot be transformed into high-level code that easy [19]. Pointers for example are just supported by a few high-level languages like C or C++. Other very important languages like Pascal, JAVA or perhaps C# do not support them [20, 5]. To eliminate these critical

elements would be a huge benefit and can be considered as necessary constraint for migration to several high-level languages [16].

Another thing that makes it difficult to migrate a legacy system is the fact that the term "legacy system" is not very well defined. Sometimes, also high-level language code without any documentation is called "legacy code". Other people consider high-level languages itself as documentation. Therewith, the migration starts from totally different stages. To avoid this problem in our work we will clearly define the properties of legacy systems we are trying to migrate. In our case a legacy system is:

- a software system or part of a software system which is implemented in an assembler language.
- a system with a very large scale and high complexity. It can e.g. consist of million lines of code.
- part of an entire legacy environment which includes computing hardware and external devices.
- a system with no or incomplete documentation or the documentation has become inconsistent during the maintenance.
- a system which is absolutely essential for the company who uses it.

The current program transformation process of the FermaT Transformation System uses a "standard transformation sequence" which cannot be customized to meet different needs and which is not optimized to achieve the best migration result [21, 14]. This sequence uses a simple strategy. It applies an atomic transformation to a specific kind of WSL code as long as there are some changes [18]. If nothing changes during a single transformation it will apply the next. The transformation sequence is not optimized and the user has no possibility to change the sequence. In other words, the sequence will be applied in a statically "try-and-error" process. If there are some special requirements on the migrated code, the user needs to change the code by hand or he can apply a single transformation manually [15]. Because of the large number of transformations and the huge scale of a typical legacy system, this process is very time-consuming and it is nearly impossible for the maintainer to find the best transformation sequence to meet his requirements [4].

The problem to find optimized transformation sequences is well known. Several approaches are existing who are describing the problem in a very good way. Also, there are some approaches to solve this problem [8, 1, 2]. But still there is the huge search space to find the optimal transformation sequence [4]. If we assume that a typical legacy assembler system consists of at least a million lines of code and we have to apply several thousand of atomic code transformations, the search space would be enormous. Also, a very efficient algorithm probably can not solve this problem. Maybe, a hierarchical system which splits the transformation sequence into sub sequences may reduce the demand of computing power [3].

The last problem is how to express a sequence? Actually, there is no language to express a transformation sequence in a meaningful way. A maintainer has no choice but to apply transformation sequences manually if he wants to customize them [17]. Also, other approaches to find transformation sequences have no possibility to express them in an abstract way [1, 8, 4].

Chapter 3

Motivation

The approach to migrate a legacy system via an intermediate language and a set of code transformations is not new [12]. It has been proven as a useful approach for software evolution [21, 19]. Nevertheless, there are some problems with this approach. One reason for these problems is the complexity of the entire program transformation process [2, 16, 21]. It consists of several thousand atomic code transformations, depending on the size of the legacy system. The FermaT Transformation Engine for example provides several dozen of mathematically proven software transformations. These transformations can be applied to the code one after another. Therewith, we have a sequence of transformations which is essential to achieve the goals of the code migration.

At each stage of the application of these rules, there are many points in a program; typically one per node of the Control Flow Graph. Therefore, the set of pairs of possible transformation rules and their corresponding application point is large [3]. Furthermore, to achieve an effective overall program transformation tactic many rules may have to be applied and each would have to be applied in the correct order to achieve the desired result. This explosion in the number of possible choice of transformation rules and their appropriate sequencing has led to problems in the generalisation of program transformation and while specialised transformation algorithms exists for dedicated tasks we are currently unaware of any general purpose transformation algorithms [4]. Also, the search-space of possible transformation sequences make exhaustive search infeasible. Given a transformation system with 20 possible code transformations and a sequence length of 20, there are 20^{20} possible combinations in the search-space which is enormous.

There have been several approaches to solve these problems but still it is not

possible for a maintainer to define constraints which are satisfying during the program transformation process. For example, Romain Robbes and Michael Lanza developed a versioning system for software evolution to analyze and optimize software evolution [9]. This systems is collecting information but is not able to take influence on the process. Deji Fatiregun, Mark Harman and Robert M. Hierons developed a system to use genetic algorithm to find suitable transformation sequences [4, 3]. They are focussing on one constraint which is "lines of code". Keith D. Cooper has developed a tool to search for optimized code transformation sequences which are used to improve the efficiency of x86 compiler [1]. Andrew P. Nisbet focused on using a genetic algorithm to find program restructuring transformations for FORTRAN programs to execute on parallel architectures [8]. All of these approaches cannot deal with more than one constraint and it is not possible to express a transformation sequence in an abstract way. Thereby, these particular functions would imply huge advantages and could help to save a lot of time and money during the migration of a legacy system.

Chapter 4

Proposal

To find suitable transformation sequences for specific requirements on the transformation result we have developed a formal language which expresses transformation sequences. This language should be powerful enough to describe complex transformation sequences but also be simple enough to be useful even for people who are not very familiar with program transformation theory. Therefore, the maintainer should be able to describe a program transformation in a very abstract way and also to control the program transformation by defining the constraints which have to be satisfied. This constraints support a priority system to avoid conflicts between them, and it is possible to group constraints to define abstract requirements. In that way the user should be able to describe e.g. his definition of maintainability. The table 4.1 shows the Backus-Nauer-Form of the formal language which is very simple at this early state of the research.

Sequence	$\langle sequence \rangle ::= \langle alternation \rangle\ (\text{","}\ \langle alternation \rangle)*$
Alternation	$\langle alternation \rangle ::= \langle repeat \rangle\ (\text{"\|"}\ \langle repeat \rangle)*$
Repeat	$\langle repeat \rangle ::= \langle option \rangle\ (\text{"+"} \mid \text{"*"})*$
Option	$\langle option \rangle ::= \langle factor \rangle\ (\text{"?"})?$
Factor	$\langle factor \rangle ::= string \mid \text{"("}\ \langle expression \rangle\ \text{")"}$

Table 4.1: This is the Backus-Nauer-Form of the constraint based transformation language.

We choose a regular language for our approach because we need to convert it into a finite automaton. This automaton visualizes the transformation process. Therewith, the user is able to observe the entire transformation process which is a huge advantage for transformation comprehension.

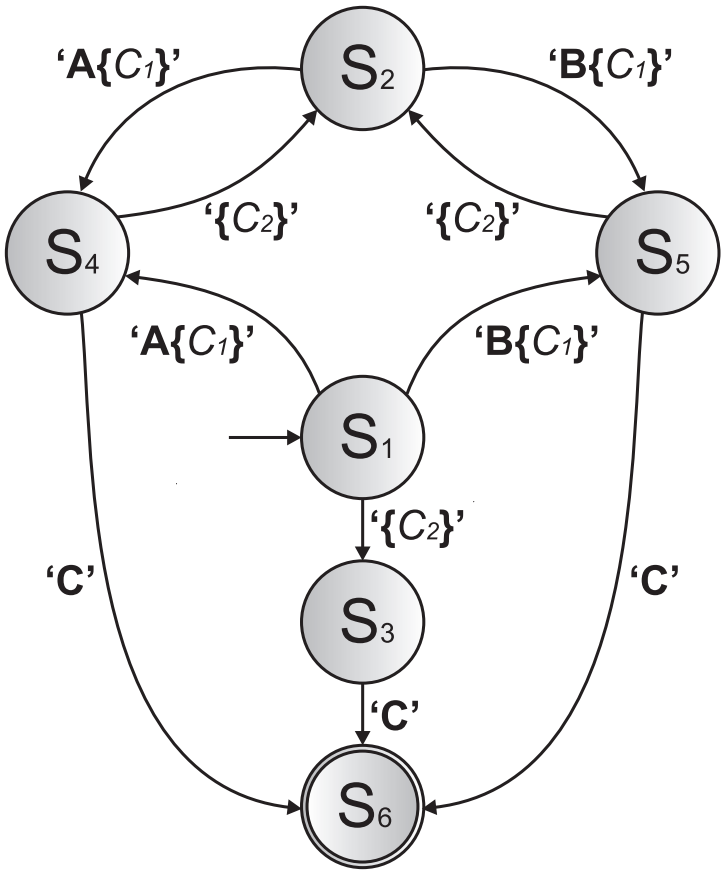

Figure 4.1: An example of an equivalent DFA D_R for the regular expression $R = (A|\{C_1\}B) * \{C_2\}, C$. This DFA includes the constraints C_1 and C_2.

A finite automaton consists of states and transitions. After we have parsed the regular language, every state of the automaton is a representation of a code version during the program transformation process. The initial state S_0 is the unchanged code which the user puts into the system. The final state S_n is the program transformation result. The transitions between the states are the atomic code transformations. Because there are several ways to travel the automaton, we need additional user information to find the best way for the requirements of the user. This will be controlled by the constraints. If there are no constraints at all, the system will return several program transformation results and the user has to choose his favourite manually. The automaton and all details of the program transformation can be shown in a graphical window. The user is able to obtain the entire transformation process and can take influence if necessary.

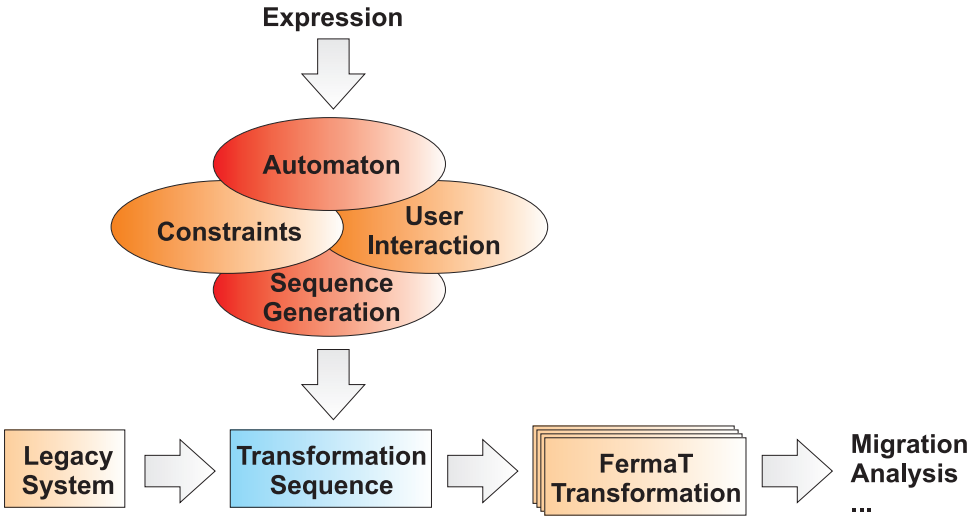

Figure 4.2: Constraint Based Transformation System.

The Constraint Based Transformation System is based on JAVA and is an extension of the FermaT Maintenance Environment. It consists of an editor and a parser for the formal language and an automaton generator to convert the language into an equivalent automaton. Furthermore, the system provides several strategies to measure WSL code and to generate transformation sequences based on user defined constraints.

Chapter 5

Progress

The following chapter will briefly discuss the progress in the research of Constraint Based Transformation Theory. It includes a literature review, a paper about Conditioned Semantic Slicing, the tool foundation of the research and initial work on a Constraint Based Transformation System. For detailed information please have a look at the appendix.

5.1 Literature Review

A major part of the work which has been achieved is a literature review about the theoretical foundation on which the research area is based. This literature review will give an overview about the principles and problems of software evolution and discuss the migration of legacy assembler systems via the "FermaT Transformation System". Furthermore, it explains the techniques and the theoretical foundation of my approach which is based on regular expressions and automaton theory.

5.2 Conditioned Semantic Slicing for Abstraction

Another achievement is a paper about the difficulties of legacy assembler analysis, comprehension and migration. In this paper we discuss the application of program transformation technology to assist with analysing and understanding legacy assembler systems. We briefly introduce the fundamentals of our program transformation theory and program slicing which generalises to conditional semantic slicing. These transformations are applied to a fairly large commercial assembler system to automatically generate high-level abstract descriptions of the behaviour of each assembler module with error handling code sliced away. The result is a dramatic improvement in the understandability of the programs: on average a

6,000 line assembler listing is condensed down to a 132 line high level language abstraction.

5.3 Progress Report on the UTC-SE Projects

Another huge part of our research and the base for the FermaT Maintenance Environment and the Constraint Based Transformation System are the FermaT tools which have been developed at the STRL in cooperation with Software Migration Limited (SML) (see appendix. These tools are developed for code measuring and re-documentation of large software systems and help to analyse the problems during a legacy system migration.

5.4 FermaT Maintenance Environment

To be able to test the proposed ideas it is necessary to develop an interface to the FermaT toolset. This has been done with the FermaT Maintenance Environment (FME). The FME is written entirely in the JAVA language. The choice for the JAVA language was made because it is a very safe and flexible language with an extensive API and a vast amount of open-source libraries for almost every possible computer task. Another advantage is the interoperability of JAVA because it is based on a virtual machine.

Primary the FME is a graphical interface to the FermaT transformation engine. It consists of a text editor which is able to express WSL together with an Abstract Syntax Tree (AST) viewer. A maintainer can navigate through the code via the source code or via the AST. The environment provides a console to the FermaT transformation engine which can be used to command the engine directly. The core of the transformation engine is a set of mathematical proven program transformation to simplify the source code. The transformations can be selected from a transformation catalogue and performed on a chunk of WSL code. The transformations will either produce a semantically equivalent or refined version of the source program construct.

This interface can be extended to process constraint based FermaT transformations and transformation sequences. The first implementation of the Constraint Based Transformation System is already integrated in the FME.

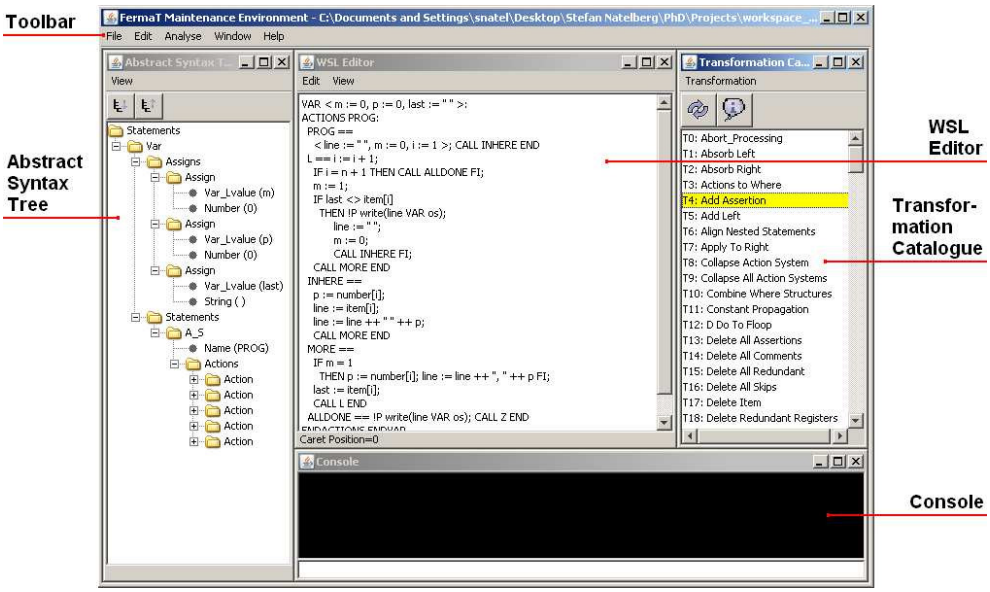

Figure 5.1: FermaT Maintenance Environment.

5.5 FermaT Documentation for Legacy Systems

FermaT Documentation for Legacy Systems (FermaT DOC) is a technology based on HTML to generate documentation of a legacy system automatically. It was developed at the STRL in cooperation with Software Migration Limited and the University of Applied Sciences Emden, Germany.

This tool was the first project of the Author at the STRL and is used as a base for further research. It provides many techniques to analyse a legacy system and to measure the quality of assembler code. Many of the ideas which came up during the development and use of FermaT DOC are implemented in the Constraint Based Transformation System.

5.6 Constraint Based Transformation System

Program transformation is used extensively in many areas of computer science including program comprehension, reverse engineering and compiler optimisation. Thereby, a program transformation usually consists of thousands of atomic code transformations which are applied to the program. We are currently working on a Constraint Based Transformation System and discusses how to find a good order in which code transformations should be applied. Furthermore, we use regular ex-

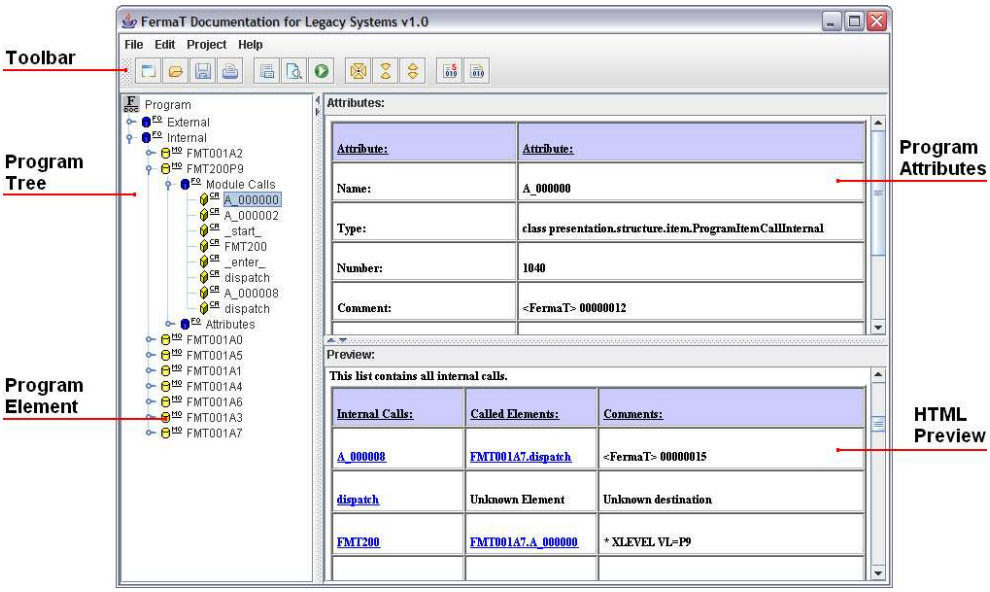

Figure 5.2: FermaT Documentation for Legacy Systems.

pressions and automaton theory to express and visualise transformation sequences.

Chapter 6

Future Work

The presented achievements have established a solid foundation for the future research. The ongoing and future work includes particularly the following work packages:

6.1 Implementation of a Program Transformation Environment

In this work package an extension for the FermaT Maintenance Environment will be developed to be able to evaluate the proposed ideas. The Program Transformation Environment will provide a graphical user interface to visualise the entire process of a program transformation. It allows the user to check different stages of the code and compares them on an abstract level. The user can take influence on the transformation process at every stage. This tool will be a proper foundation for further research.

6.2 Formal Language for Constraint Based Program Transformation

In this work package the language for constraint based program transformations will be defined. Therefore, we will also implement an editor and a parser to generate automata which will represent the transformation process.

6.3 Constraint Based Transformation System

In this work package we will design and implement the Constraint Based Transformation System. It is based on the Program Transformation Environment and provides several strategies to meet constraints, analyse migration results and evolve transformation sequences for different requirements.

6.4 Test of the Approach using large scale case studies

In this package the possibilities of the approach will be demonstrated on medium to large scale case studies.

6.5 Thesis writing up

The last work in this research project will be the documentation of all outcome into a PhD thesis.

Chapter 7

Thesis Outline

The following describes the proposed outline of the author's PhD thesis. The thesis will concentrate on constraint based transformation theory issues and shows how a transformation sequence can be expressed uniformly within an expression language. The structure of the thesis is envisioned as follows:

Chapter 1: Introduction

Chapter 2: Preliminaries

- Literature Review on Transformation Theory.
- Literature Review on Automaton Theory.
- Literature Review on Formal Languages.

This chapter will give an introduction of the related research areas.
Completed: 80%, Estimated Time: 5 Weeks

Chapter 3: Related Work

- The FermaT Transformation Theory.
- Evolving Transformation Sequences.

This chapter will give an overview about related work in this research area.
Completed: 80%, Estimated Time: 7 Weeks

Chapter 4: A Formal Language for Constraint Based Program Transformation

- Definition of constraints for program transformation.
- Development of a formal language to express transformation sequences.

This chapter will discuss the base for further research.
Completed: 45%, Estimated Time: 15 Weeks

Chapter 5: A Constraint Based Transformation System

This chapter will discuss theoretical foundation of a Constraint Based Transformation System.
Completed: 20%, Estimated Time: 10 Weeks

Chapter 6: Implementation

- Design and implementation of the FermaT Maintenance Environment.
- Design and implementation of the Constraint Based Transformation System.

This chapter will discuss the implementation of the FermaT Maintenance Environment and the Constraint Based Transformation System.
Completed: 40%, Estimated Time: 6 Weeks

Chapter 7: Case Studies and Evaluation

This chapter will disuss two or more medium to large scale case studies.
Completed: 10%, Estimated Time: 4 Weeks

Chapter 8: Conclusion and Future Work

This chapter will summarise the research outcome and outline future work.
Completed: 0%, Estimated Time: 2 Weeks

Bibliography

[1] K. D. Cooper, P. J. Schielke, and D. Subramanian, editors. *Optimising for reduced code space using genetic algorithms*. Proceedings of the 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES), May 1999.

[2] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimising compilers for the 21st century. *Journal of Super Computing*, 23:7–22, 2002.

[3] D. Fatiregun, M. Harman, and R.M. Hierons. Search based transformations. In *Genetic and Evolutionary Computation Conference*, Chicago, USA, July 2003.

[4] D. Fatiregun, M. Harman, and R.M. Hierons. Evolving transformation sequences using genetic algorithms. In *Department of Information Systems and Computing, Brunel University*, 2004.

[5] Jay Hilyard and Stephen Teilhet. *C Sharp Cookbook*. O'Reilly Media, February 2006.

[6] Kapsu Kim, Yeongil Shin, and Chisu Wu. Complexity measures for object-oriented program based on the entropy. Technical report, Seoul National University - Department of Computer Science & Statistics, 1995.

[7] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68:1060–1076, September 1980.

[8] A.P. Nisbet. Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation. *HPCN Europe*, 1:987–989, 1998.

[9] R. Robbes and M. Lanza. Versioning systems for evolution research. *Principles of Software Evolution*, 1:155–164, September 2005.

[10] Ladan Tahvildari, Richard Gregory, and Kostas Kontogiannis. An approach for measuring software evolution using source code features. Technical report, University of Waterloo - Department of Electrical & Computer Engineering, 1999.

[11] M. Ward. Transforming a program into a specification. Technical report, Durham University, January 1988.

[12] M. Ward. *Proving Program Refinements and Transformations*. PhD thesis, Oxford University, 1989.

[13] M. Ward. Abstracting a specification from code. *Journal of Software Maintenance: Research and Practice*, 5:101–122, 1993.

[14] M. Ward. Assembler to c migration using the fermat transformation system. In *International Conference on Software Maintenance, Oxford*, August 1999.

[15] M. Ward. The fermat assembler re-engineering workbench. In *International Conference on Software Maintenance, Florence*, 2001.

[16] M. Ward. Pigs from sausages? reengineering from assembler to c via fermat transformations. *Science of Computer Programming*, 52:213–255, 2004.

[17] M. Ward, F. Callis, and M. Munro. The maintainer's assistant. In *Conference on Software Maintenance, Miami Beach, Florida*, 1989.

[18] M. Ward and T. Hardcastle. *WSL Programmer's Reference Manual*, November 2003.

[19] M. Ward and H. Zedan. Analysing and abstracting legacy assembler code via conditioned semantic slicing. Technical report, Software Technology Research Laboratory, De Montfort University, 2006.

[20] Karsten Wolke, Anja Austermann, and Kostyantyn Yermashov. *Das Java 2 JDK 5 Lehrbuch*. Computer & Literatur, 2005.

[21] H. Yang and M. Ward. *Successful Evolution of Software Systems*. Artech House Publishers, January 2003.