Automating Software Evolution: Towards Using Constraints with Action for Model Evolution

By

Shahid Alam, B. Eng., M. Sc.

A Thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the
requirements for the degree of

Master of Applied Science
in
Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University, Ottawa, Ontario
© Copyright Shahid Alam 2007
The undersigned hereby recommends to the faculty of Graduate Studies and Research acceptance of the thesis

**Automating Software Evolution: Towards Using Constraints with Action for Model Evolution**

Submitted by Shahid Alam, B. Eng., M. Sc.
in partial fulfillment of the requirements for the degree of

**Master of Applied Science**
in
**Electrical Engineering**

______________________________
Professor Samuel A. Ajila
Thesis Supervisor

______________________________
Professor Victor C. Aitken
Chair, Department of Systems and Computer Engineering

Carleton University
2007
Abstract

In this thesis we examine two issues and propose solutions to resolve them. The first is the automation of model evolution and the second is the support of software evolution in modeling languages. We extend Object Constraint Language with actions and define a new language CAL (Constraints with Action Language), which gives a user the ability to use constraints with actions on models. We have added a data type, directed acyclic graph (DAG) to CAL to automate model evolution and optimize impact analysis, and to make CAL more accessible to non-programmers. The DAG has been annotated (ADAG) with dependency weights and a labeling scheme has been improved in size and used to optimize basic ADAG operations. CAL contains a small set of constructs, but is powerful enough to be used efficiently for typical software evolution management operations. A prototype tool VCal, for dependency analysis of UML Class Diagrams is presented.
Acknowledgements

I would like to acknowledge the contributions of Dr. Samuel A. Ajila to the writing of this thesis in the form of content, resources, commitment and support. I would specially like to mention his encouragement and guidance, for writing research papers and becoming a part of the software engineering research community.

Numerous other people deserve to be mentioned for their parts in my Master’s thesis. I would like to thank Professor Samuel A. Ajila and Professor Halim Yanikomeroglu and other staff members of the department for their advice and support during my master’s program. The Graduate Communication Workshop by Professor Logie, and Kate Parsons of the Academic Writing Centre provided guidance and feedback in writing this thesis.

Special thanks go to my family: my parents, wife and four kids for their understanding, support and help, which enabled me to accomplish this.
**Acronyms**

ADAG = Annotated Directed Acyclic Graph  
ADP = Acyclic Dependency Principle  
AOP = Aspect Oriented Programming  
AS = Action Semantics  
ASL = Action Specification Language  
AST = Abstract Syntax Tree  
AWT = Abstract Window Toolkit  
BFS = Breath First Search  
CAL = Constraints with Action Language  
CASE = Computer Aided Software Engineering  
CFG = Context Free Grammar  
DFS = Depth First Search  
EBNF= Extended Backus-Naur Form  
ECL = Embedded Constraint Language  
GUI = Graphical User Interface  
MDA = Model Driven Architecture  
MDSE = Model Driven Software Engineering  
OAL = Object Action Language  
OCL = Object Constraint Language  
OMG = Object Management Group
PAL = Platform Independent Action Language

PIM = Platform Independent Model

PSM = Platform Specific Model

RFP = Request for Proposal

SWT = Software Widget Toolkit

TLA = Temporal Logic of Action

UML = Unified Modeling Language

VCAL = Visual CAL
Contents

ABSTRACT III

ACKNOWLEDGEMENTS IV

ACRONYMS V

INTRODUCTION 1

1.1 Motivation for CAL 2

1.2 Problem statement 3

1.3 Goals 5

1.4 Approach used 6

1.5 Research contributions 9

1.6 Organization of this thesis 11

BACKGROUND AND MODEL EVOLUTION 12

2.1 Software evolution 12

2.1.1 Issues in software evolution 13

2.1.1.1 Raising the level of abstraction 13

2.1.1.2 What, how and why? 13

2.2 Dependency relationships between UML model elements 14

2.3 Formal languages 14

2.3.1 Specification of formal languages 17

2.3.2 Left recursion removal in LL(k) grammar 18

2.3.3 Left factoring 18

2.4 Executable UML 19

2.5 OCL (Object Constraint Language) 20

2.5.1 OCL collections 22

2.5.2 OCL navigation 22

2.5.3 OCL standard library 22
2.6 TLA (Temporal Logic of Actions) 23
  2.6.1 An algorithm language (pluscal) based on TLA 25

2.7 Related work in model evolution 29
  2.7.1 Evolution contracts 29
  2.7.2 Refactoring 30
  2.7.3 Model evolution and impact analysis 31
  2.7.4 Recent languages for model transformation 33
  2.7.5 Comparison 35
  2.7.6 Impact analysis support in commercial UML tools 38

CAL 41

3.1 CAL grammar 42
  3.1.1 Production rules added for CAL 43

3.2 CAL library 45
  3.2.1 DagType 45
  3.2.2 Dag 46
  3.2.3 isdag 47
  3.2.4 size 47
  3.2.5 isempty 47
  3.2.6 flatten 47
  3.2.7 includes 48
  3.2.8 excludes 48
  3.2.9 depend 48
  3.2.10 descendants 49
  3.2.11 view 50

3.3 CAL actions 50
  3.3.1 create 51
  3.3.2 delete 51
  3.3.3 link 52
  3.3.4 unlink 52
  3.3.5 addattr 53

3.4 Usage 53

3.5 CAL and other action languages 55

DAG FOR DEPENDENCY ANALYSIS 61

4.1 DAG Labeling 64
  4.1.1 Labeling scheme 66
  4.1.2 Size of the label 68

4.2 Labeling and dependency analysis algorithms 69
4.2.1 Time complexity
4.3 Updates
4.4 Related work
CAL IMPLEMENTATION
5.1 Rules for translating a UML model to the CAL data structure
5.2 CAL model
  5.2.1 ADAG operations
    5.2.1.1 Includes
    5.2.1.2 Excludes
    5.2.1.3 Depend
    5.2.1.4 Descendants
  5.2.2 CAL actions
    5.2.2.1 Create
    5.2.2.2 Relabel
    5.2.2.3 Setweight
5.3 Formal verification using TLC
CAL PROTOTYPES
6.1 CAL parser
6.2 Implementation details
  6.2.1 Class Parser
  6.2.2 Class OclStmt
  6.2.3 Class Lexer
  6.2.4 Class AST
    6.2.4.1 Data structure for storing the AST
  6.2.5 Class Symtable
    6.2.5.1 Collision resolution
      6.2.5.1.1 Chaining
      6.2.5.1.2 Open addressing
  6.2.6 Class LG
  6.2.7 Class List
6.3 VCAL (visual CAL)
  6.3.1 CAL parser and CAL file
  6.3.2 ProcessXMI
  6.3.3 ADAG (annotated DAG)
  6.3.4 CAL Interpreter
  6.3.5 GUI (graphical user interface)
  6.3.6 Running VCAL
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Stereotypes for dependency in UML 2.0</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Dependency relationships between UML model elements</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Current OCL tools</td>
<td>40</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison of CAL with other action languages</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Characteristics and coupling of different relationships in UML</td>
<td>81</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparison of LL and LR parsing</td>
<td>97</td>
</tr>
<tr>
<td>6.2</td>
<td>Chaining versus open addressing</td>
<td>103</td>
</tr>
<tr>
<td>6.3</td>
<td>Test matrix for the case study</td>
<td>112</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Overview of the solution proposed in this thesis.................................................. 10
1.2 Overview of the CAL model verification............................................................. 10
2.1 Simple Clock Example from the TLA book [50].................................................... 26
2.2 Euclid Algorithm in pluscal................................................................................. 28
3.1 Production rules for CAL....................................................................................... 44
3.2 Sample CAL file.................................................................................................... 54
4.1 Sample DAG.......................................................................................................... 65
4.2 Equations for computing the label of a vertex in the DAG................................. 67
4.3 Labeled DAG........................................................................................................ 68
4.4 Algorithms to label a node in the DAG............................................................... 70
4.5 Algorithm to find descendants of a node in the DAG............................................ 71
4.6 Algorithm to find reachability of two nodes in the DAG...................................... 72
4.7 Algorithms to remove the link of a node from the DAG, and check if a node is
    present or not in the DAG.................................................................................... 73
4.8 Algorithm to set the parent of a node in the DAG............................................... 74
4.9 Algorithms to add and delete a node in the DAG................................................. 75
4.10 Labeled DAG with node R Added....................................................................... 76
4.11 Labeled DAG with node K Removed.................................................................. 77
5.1 Global variables (data) and define block.............................................................. 83
5.2 ADAG operation includes..................................................................................... 84
5.3 ADAG operation excludes.................................................................................... 84
5.4 ADAG operation CheckDependency 85
5.5 ADAG operation depend 86
5.6 ADAG operation descendants 87
5.7 CAL action create 89
5.8 CAL sub-action relabel 90
5.9 CAL sub-action setweight 91
5.10 Part of the main block of CAL specifications, and the CAL data structure (ADAG) before and after annotations 92
5.11 Part of the TLC output 93
6.1 Data structure for storing AST 99
6.2 XML output generated by the Class AST 100
6.3 Collision resolution by chaining 102
6.4 Collision resolution by open addressing 103
6.5 Linked List with pointer to the data 105
6.6 Overview of the tool (VCAL) 105
6.7 VCAL GUI 108
6.8 UML Class Diagram for the CAL parser in XMI 110
6.9 CAL files, with different CAL actions and operations 113
6.10 VCAL GUI, showing DAG after executing different CAL actions and operations shown in Figure 6.9 114
Chapter 1

Introduction

Software evolution has become a consistent and major part of the software life cycle. More time is spent on changing than developing the software. Of all the software processes evolution is the most expensive. There is a need to evolve the system from the beginning. To achieve this we want to integrate software evolution in the design or the model of the software system. Since the advent of model driven software engineering (MDSE) it has become more necessary to develop techniques and tools for model evolution. Due to the lack of support of software evolution management in modeling languages and tools, it is highly desirable to integrate such support in these modeling languages and tools. None of the previous efforts, either commercial [24, 25, 26, 33, 34, 35] or academic [12, 14, 15, 19, 20, 22, 28], support automation of software evolution in a modeling language. This thesis proposes a novel approach for the automated management of software evolution in models. The basic purpose of this approach is to show that a modeling language can be used to automate software evolution in models.

In this thesis, we extend OCL (Object Constraint Language) and define a new language CAL (Constraints with Action Language). We have added 5 actions, 10 operations and 1 data type to CAL. OCL is a query modeling language [2] used for the models specified in a Unified Modeling Language (UML) [5] and can also be used for other models. OCL can be used to define constraints (rules) for impact analysis [22], but
the use of constraints is not the optimal way to perform impact analysis, and moreover OCL cannot be used to make changes to a model. CAL gives the user the ability to use constraints with actions on models. Because of the characteristics (transitive and non-reflexive) of each dependency relationship between UML Class Model elements, a new set of Collection directed acyclic graph (DAG) has been added to the CAL library for automatically generating these dependency relationships. The DAG has been annotated for dependency analysis, with dependency weights and an improved labeling scheme has been proposed and used to optimize basic DAG operations (also called CAL operations). We call this DAG an annotated DAG (ADAG).

CAL contains a small set of constructs, but is powerful enough to be used efficiently for typical software evolution management operations like impact analysis, correction, improvement and enhancement of models. CAL can be extended and used, as an action language, or in model compilers for incremental recompilation and retransformation of models, in model driven architecture (MDA) [11, 47]. Since CAL has its own well defined data structure and type system, it is independent of other modeling tools API's, data structure and type system. This makes CAL highly interoperable, portable and flexible.

1.1 Motivation for CAL

There are two important concepts in software engineering, use of models and automation in software engineering, on which this thesis is based. Separating the model of the software system from its implementation allows us to integrate software evolution in the design of the software system. Software models are easier to maintain, evolve and understand. Empirical investigation shows that even experienced software professionals
predict incomplete sets of software changes [45]. One of the challenges [44] facing software evolution is to raise the level of abstraction and is called model evolution.

Introduction of MDA by object management group (OMG) has opened up new opportunities in model evolution. MDA introduces the concept of executable models, to raise the level of abstraction. Executable models are computationally complete models and therefore can be executed [9, 16]. The implementation follows the model and not vice versa and hence we only need to change the model. MDA is an important software design approach towards MDSE, creating new opportunities for model evolution. There are two issues that our work examines and resolves. One of them is the support of software evolution in modeling languages. There are many computer aided software engineering (CASE) tools including commercial, open source and academic that are being used in MDSE. Automation of the software development life cycle is at the heart of these CASE tools. The other opportunity is the automation of model evolution.

1.2 Problem statement

Impact analysis is defined as identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change [7] and is one of the methods used in managing software evolution. Although it is easier to collect the necessary information for impact analysis at detailed design level than at the source code level, still impact analysis of models faces problems. Some of these problems, which this thesis examines, are listed below:

- Lack of support of management of software evolution in modeling languages

UML is a well-established industrial standard for modeling software systems and is becoming more mature as time passes. It has notations, symbols and stereotypes to
define dependency relationships, but there is no modeling technique for automated software evolution management. OCL or other languages [24, 25, 26] do not directly provide support for model evolution.

- **Automation of the management of software evolution in models**

There are several reasons for change and it is not easy to identify the impact of change in a software system [7]. Dependency relationships, whether direct or indirect between UML model elements are numerous. To find them by hand is not practical as it involves lot of labor and hence is prone to errors. It is highly desirable to develop techniques to automate this process to reduce errors, time and money especially for large and complex systems. After identifying these dependency relationships, how can we automatically perform typical software maintenance operations like correction, improvement and enhancement to the model? There are various efforts at detecting, tracing and managing evolution but very few [14, 18, 22] deal with automating this process and none cover all the aspects mentioned above.

- **Lack of support of automated management of software evolution in tools**

Impact analysis of the model should be part of a good CASE tool. We not only need support of tools but we need them to automate this process with as little input from the user as possible. Currently there is lack of support of impact analysis in CASE tools. Very few CASE tools provide this facility either by themselves through a set of procedural application programming interfaces (API's) [32, 33, 34] or by third party plug-ins such as Gateway [35] in Rhapsody. If it is provided then it is highly dependent on the CASE tool and requires great amount of effort from the user, so this indirect support is not portable, flexible and automatic.
• Incremental recompilation and retransformation of models

MDA uses model transformation language to transform model from platform independent model (PIM) to platform specific model (PSM). Executable UML [9, 16] takes the concept of MDA further. It removes the PSM from MDA and directly generates code that can execute. How can change be propagated automatically to the transformed model or to the generated code, so that when the software modeler makes a change to the model, the model compiler is able to detect which elements of the model have been modified and then use an impact algorithm to determine what minimal elements, need to be recompiled or retransformed, and in what order?

1.3 Goals

We desire a practical solution that gives us the ability to automate the process of generating the dependency relationships among the model elements for impact analysis. Besides being automatic it should be visual, portable, efficient, flexible and provide ease of use. Following is a description of these characteristics in the context of this thesis:

● Portability

The solution should be portable to other CASE tools, which means it should be easily integrated into other CASE tools and makes it more reusable. It should be independent of any domain (like hardware, embedded systems etc).

● Flexibility

It is highly desirable for a software evolution system to do updates. It should be able to update the changes made in the model. Update here, means updating the changes in the model without going back to the original model. In this way, there is no need to compare two models or to keep an extra model in the working
memory before making a change. Besides updates the solution should be scalable
to accommodate future extensions.

- **Efficiency**
  The queries for finding dependencies among model elements should take less
time. In an ideal case the time should be constant i.e. $O(1)$ time to complete. Since
flexibility impacts on efficiency, it needs to compensate for that while updating.

- **Ease of use**
  A CASE tool or any tool is judged by its ease of use. It should require least
amount of input from the user for generating impact analysis results. The user
should be able to interact with the results.

- **Visualization**
  Visual or graphical presentation of impact analysis of software change is highly
desirable in a CASE tool. Graphical abstraction helps us focus on the
relationships among elements of the model that are being maintained.

Besides these characteristics we would like to see if the solution or part of the
solution has a mathematical model and uses well defined standard notations to describe
its rules.

### 1.4 Approach used

In this section we define an approach that is used in solving the problems and
achieving the goals mentioned above. OCL is basically used as a query language for
UML. Expressions such as constraints and rules in OCL can be used to identify changes
[22], but cannot be used to change the state of the system [2]. Therefore, OCL alone
cannot be used for making changes to the model. Action semantics [27] is another OMG
standard and is now part of UML 2.0 that is specifically defined for this purpose. We have extended OCL with actions and defined a new language CAL. OMG does not define any syntax or semantics for the actions, so we have formally defined the syntax and semantics of CAL actions. A new collection library DAG has been added to automatically generate dependency relationships, which can be queried and elements can be added or deleted using different library functions. Dependency weights have been added to the DAG for dependency analysis. These weights are based on the coupling of different dependency relationships between UML Class Model elements. This gives us a powerful way of querying for dependencies and changing and manipulating UML Class Model elements to achieve automated software evolution management of UML Class Models. CAL is well defined with a grammar in EBNF [1] and an efficient LL(1) parser implemented in C++ for this thesis. CAL is based on OCL and action semantics, which make this solution highly portable and flexible as defined above.

An understanding of dependency relationships between model elements is necessary for efficient impact analysis of models. First we distinguish all the dependency relationships between UML Class Model elements. Then we store these dependencies in a dependency graph. Each vertex represents a UML Class Model element and each edge represents a dependency relationship between the elements. Each vertex is assigned a weight as described above for dependency analysis. The characteristics of each dependency relationship are transitive and non-reflexive. We also assume a good object oriented design based on the Acyclic Dependencies Principle (ADP) [46] i.e. no cycles between model elements. Therefore, the dependency graph by definition is a DAG. The rules of translation from a UML Class Model to the DAG have been formally specified
for better comprehension. As mentioned above one of the CAL libraries gives user the ability to query, add or delete elements from the DAG. In this way we can perform all the tasks of impact analysis [7] using the DAG. As executable UML comprises the Class Model, the State Machines for the classes and the state procedures, so we restrict our dependency relationships to Class Diagrams for the prototype tool VICAL (Visual CAL), which is implemented for this thesis, but the approach can be used for other UML elements, which can be developed independently of other elements and satisfies ADP.

An improved labeling scheme is proposed for labeling the DAG to achieve efficient query operations for finding reachability between two elements and descendants of an element and add and delete operations. Reachability means that the two elements are dependent and descendants of an element are a set of elements that depend on this element. Both of these can be used to analyze the impact of changing the element. Each element's dependency weight can be calculated by adding all its descendant’s weight, which further helps in analyzing the dependencies and the impact of change. The prototype tool VICAL implemented for this thesis checks for reachability between two or more than two elements and descendants of an element.

After adding weights and labeling the DAG we call it an annotated DAG (ADAG). For visualization of the ADAG we use Java graphics visualization and layout library JGraph [48], which was easy to implement for a research based prototype solution. But for commercial purposes we recommend C++ with OpenGL [49], which is being used in commercial CASE tools for simulations like Simulink and Matlab. Standard widget toolkit (SWT) [54] has been used for design and development of the GUI. It is a Java class library that allows the user to create native user interfaces. It is
designed to provide efficient, portable access to the underlying facilities of the operating system on which it is implemented. SWT implementation uses Java native interface (JNI) [55] to accomplish this. Figure 1.1 shows an overview diagram in UML of the components of the prototype tool VCAL implemented as part of the solution. We have tested and verified this tool using the CAL parser UML class diagram as an example case study.

1.5 Research contributions

Following are the research contributions of this thesis:

✓ There are two issues that our work examines and resolves. One of them is the support of software evolution in modeling languages. The other issue is the automation of model evolution.

✓ A solution is provided in the form of a modeling language CAL (Constraints with Action Language). An overview of the solution is shown in Figure 1.1. CAL gives a user (modeler) the ability to use constraints with actions on models, and provides the user with tools to automate model evolution.

✓ A new set of Collection DAG has been added to the CAL library. The DAG has been annotated (ADAG) with dependency weights for dependency analysis, and a labeling scheme has been improved and used to optimize basic ADAG operations (also called CAL operations).

✓ We have also formally defined rules to translate a UML Class Model to the CAL data structure (ADAG).

✓ The CAL model, including all the CAL operations and their libraries and one of the CAL actions, create, and its sub-actions, relabel and setweight, have been
formally specified using *pluscal* [53], which is based on Temporal Logic of Actions+ (TLA+) [50], and is verified using TLC [50], a TLA+ model checker (see Figure 1.2).

✓ A prototype tool, VCAL, has been developed to show how CAL can be used to automatically generate dependency relationships among model elements of UML Class Diagrams to perform typical software evolution management operations (like impact analysis, correction, improvement and enhancement) with as little input from the user as possible. VCAL also visualizes and applies basic filters to the ADAG.

![Diagram](image)

**Figure 1.1** Overview of the solution proposed in this thesis

**Figure 1.2** Overview of the CAL model verification
1.6 Organization of this thesis

This chapter gives an introduction, discusses motivation and briefly describes specific issues that this thesis examines. It sets up some goals and then provides an approach that is used to solve the issues and achieve these goals.

Chapter 2 provides a background study, which aids in understanding CAL, its formal specifications, parser and the techniques used in implementing CAL, and a literature review on model evolution. At the end of chapter 2 we also compare two of the recent approaches with CAL.

Chapter 3 describes CAL in detail, including its actions, operations and data structure, and also compares it with other action languages for software models.

Chapter 4 discusses in detail the CAL data structure (ADAG) and proposes an improved labeling scheme for the ADAG for dependency analysis.

In chapter 5 we formally specify rules for translating a UML Class Model to the CAL data structure, and all the CAL operations, and one of the CAL actions, create, and its sub-actions, relabel and setweight using TLA+. Chapter 5 also describes the testing and verification of these specifications using TLC.

Chapter 6 describes the CAL parser and its implementation in detail. It also briefly explains and tests the prototype tool VCAL using a case study, which is the UML model of the CAL parser itself.

Chapter 7 concludes the thesis and highlights research contributions of this thesis and future research possible with CAL.
Chapter 2

Background and Model Evolution

This Chapter is divided in two parts. The first part provides background on software evolution, dependency relationships between Unified Modeling Language (UML) model elements, formal languages [3, 6], executable UML [9, 16], object constraint language (OCL) [2] and temporal logic of actions (TLA) [50], which aids in understanding constraints with action language (CAL), its formal specifications, parser and the techniques used. The second part describes the state of the art in model evolution, starting from evolution contracts [28] by Tom Mens and Theo D’Hondt in the year 2000, to the most recent efforts on model evolution and impact analysis. At the end a comparison is made between two of these efforts with CAL.

2.1 Software evolution

Biological species evolve for various reasons and one of them is to better cope with their environments. Similarly there are various reasons why software evolves and needs maintenance. We can say that software evolution is part of software maintenance and in this thesis we do not differentiate between software evolution and software maintenance. Of all the software development processes evolution is the most expensive. Hence, it needs more research and practice. The Research Institute in Software Evolution (RISE) [61] formally defines software evolution as the following:

“The set of activities, both technical and managerial, that ensures that software
continues to meet organisational and business objectives in a cost effective way”.

2.1.1 Issues in software evolution

The European Research Consortium for Informatics and Mathematics (ERCIM) and European Science Foundation (ESF) jointly organized a workshop, Challenges in Software Evolution (ChaSE 2005) [44], in April 2005. The aim of the workshop was to identify substantial obstacles to software evolution research and practice and to propose and discuss challenges in software evolution. The top 5 challenges in the list composed by this workshop are: (1) theory of software evolution, (2) raising the level of abstraction, (3) tool complexity, (4) language support and (5) process support. We discuss challenge number (2) below:

2.1.1.1 Raising the level of abstraction

Software is more than the source code. Models and metamodels play a key role in software evolution [52]. Software itself does not evolve but our understanding about it changes. Hence, the models that define the structure of the software change. This means that the models are easier to maintain, evolve and understand.

2.1.1.2 What, how and why?

Since the advent of model driven software engineering it has become more necessary to develop techniques and tools to overcome this challenge of raising the level of abstraction in software evolution. Model driven architecture (MDA) allows us to separate platform independent model (PIM) from platform specific model (PSM). To take full advantage of MDA, we need more work and research on model evolution. Executable UML takes the concept of MDA further. It removes the PSM from MDA and
directly generates code that can execute. To support executable UML, which is gaining popularity, we need to fully introduce evolution at the model level. Furthermore we must be able to answer the following questions. How can evolution affect and help code generation in executable UML? How can we implement a framework that can detect, trace and manage evolution in models and how can we support tools for managing evolution at this level? How can we automate model evolution? It is possible by introducing a model manipulation language. But what else do we need in that language for model evolution? Do we need to extend UML or use the existing UML specifications? It is easier to collect the necessary information, such as, the number of classes used, their operations and attributes etc, at the design level than at the source code level. How do we address change, including correction, improvement and enhancement at this level of abstraction?

2.2 Dependency relationships between UML Class Model elements

In UML, dependency relationships can be divided into four different relationships. These dependency relationships are described in Table 2.2. Many standard stereotypes for UML 1.x are now obsolete in UML 2.0. Table 2.1 lists all the stereotypes defined in [5] for dependency relationships in UML 2.0.

2.3 Formal languages

A language contains a collection of words or collection of sentences that can be formally defined by mathematical formulas and can be processed by a machine. If $F$ is a finite length of sequences of elements drawn from a finite set $S$ of symbols, then we can define a formal language $L$ mathematically as follows:

$$L = \{ S, F \}$$
if $S = \{ a, b \}$

then $F$ can be $abbabbabb$ or $abbabb$

<table>
<thead>
<tr>
<th>Name</th>
<th>Metaclass</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;call&gt;&gt;</td>
<td>Usage</td>
<td>A call dependency specifies that an operation in the source class invokes an operation in the target class.</td>
</tr>
<tr>
<td>&lt;&lt;create&gt;&gt;</td>
<td>Usage</td>
<td>A usage dependency denoting that the client classifier creates instances of the supplier classifier.</td>
</tr>
<tr>
<td>&lt;&lt;drive&gt;&gt;</td>
<td>Abstraction</td>
<td>A derived dependency specifies that the client may be computed from the supplier.</td>
</tr>
<tr>
<td>&lt;&lt;import&gt;&gt;</td>
<td>Permission</td>
<td>A package import is a relationship that allows the use of unqualified names to refer to package members from other namespaces. &lt;&lt;import&gt;&gt; is public and &lt;&lt;access&gt;&gt; is private.</td>
</tr>
<tr>
<td>&lt;&lt;access&gt;&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;instantiate&gt;&gt;</td>
<td>Usage</td>
<td>A usage dependency among classifiers indicating that operations on the client create instances of the supplier.</td>
</tr>
<tr>
<td>&lt;&lt;merge&gt;&gt;</td>
<td>Relationship</td>
<td>A package merge defines how the contents of one package are extended by the contents of another package. It also defines package merge rules if certain elements in both packages represent the same entity.</td>
</tr>
<tr>
<td>&lt;&lt;refine&gt;&gt;</td>
<td>Abstraction</td>
<td>Specifies a refinement relationship between model elements at different semantic levels, such as analysis and design.</td>
</tr>
<tr>
<td>&lt;&lt;send&gt;&gt;</td>
<td>Usage</td>
<td>A usage dependency whose source is an operation and whose target is a signal, specifying that the source sends the target signal.</td>
</tr>
<tr>
<td>&lt;&lt;substitute&gt;&gt;</td>
<td>Substitution</td>
<td>A substitution is a relationship between two classifiers which signifies that one Classifier complies with the contracts specified by the other classifier.</td>
</tr>
<tr>
<td>&lt;&lt;trace&gt;&gt;</td>
<td>Abstraction</td>
<td>Specifies a trace relationship between model elements. Traces are mainly used for tracking requirements and changes across models.</td>
</tr>
<tr>
<td>&lt;&lt;use&gt;&gt;</td>
<td>Usage</td>
<td>A usage dependency in which one element requires another element (or set of elements) for its full implementation or operation.</td>
</tr>
</tbody>
</table>
Table 2.2 Dependency relationships between UML Class Model elements [5]

<table>
<thead>
<tr>
<th>Dependency Relationship</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dependency</strong>: Dependency is a relationship that signifies that an element requires another element for its specification or implementation. There is a supplier and client relationship between model elements, where the modification of the supplier may impact the client.</td>
<td><img src="#" alt="Dependency Diagram" /></td>
</tr>
<tr>
<td>NamedElement–1 is dependent on NamedElement–2. The stereotype &lt;&lt;dependencyName&gt;&gt; can be any name from Table 2.1.</td>
<td></td>
</tr>
<tr>
<td><strong>Generalization</strong>: Generalization is a relationship between a more general classifier and a more specific classifier, and is owned by the specific classifier. Each instance of the specific classifier is also an instance of the general classifier. Generalization hierarchies are directed and acyclic.</td>
<td><img src="#" alt="Generalization Diagram" /></td>
</tr>
<tr>
<td>Shape is a general class, for Square, Circle and Rectangle, which are specific classes.</td>
<td></td>
</tr>
<tr>
<td><strong>Realization</strong>: Realization is a specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other representing an implementation of the latter (the client).</td>
<td><img src="#" alt="Realization Diagram" /></td>
</tr>
<tr>
<td>Business class is the supplier; Owner and Employee classes are its clients.</td>
<td></td>
</tr>
<tr>
<td><strong>InterfaceRealization</strong>: InterfaceRealization is a specialized realization relationship between a classifier and an interface. It implies that the classifier supports the set of features owned by the interface, and any of its parent interfaces.</td>
<td><img src="#" alt="InterfaceRealization Diagram" /></td>
</tr>
<tr>
<td>The interface ISensor is the provided interface for ProximitySensor and the required interface for TheftAlarm.</td>
<td></td>
</tr>
<tr>
<td><strong>Composition</strong>: Composition is a form of physical aggregation. It has a strong ownership between part and whole over their lifetime. It is distinguished from regular aggregation by a filled diamond. The multiplicity at the aggregate should not exceed 1. The parts depend on the aggregate. They are destroyed with the deletion of the aggregate.</td>
<td><img src="#" alt="Composition Diagram" /></td>
</tr>
</tbody>
</table>
2.3.1 Specification of formal languages

As defined above $L$ is a set of strings over a set of symbols. It is desirable to consider languages that contain an arbitrary number of strings. We do not want to put an upper bound on the number of strings, so these languages cannot be specified by enumerating the set of strings in the language. Hence, we want our specification of the language to be finite although the language being specified is not finite. There are several techniques for language specification that fulfill this requirement. One of the techniques is to use a generative system known as grammar. A Grammar is a mathematical system for defining a language and a method for giving sentences in the language a useful structure. A grammar consists of a set of terminals, a set of non-terminals, a set of production rules and a starting symbol. Production rules are the heart of a grammar and are used to generate the sentences of the language. There are many grammar forms but most of the formal languages can be defined by a context free grammar (CFG) [6], and we have used CFG to define CAL. Another technique is to use a procedure that when given an arbitrary input string will halt and answer yes if the given input string is in the language and vice versa. This procedure will take a finite amount of time to compute the answer and is called a recognizer for the language. We will now look into the first technique and briefly describe the two most popular grammars used for specifying formal languages.

LL(k) grammar. This is the grammar for which left parsers (also called top-down parsers) [3] can be built deterministically if the parser is allowed to look $k$ input symbols (tokens) to the right of its current input position. Poorly designed languages usually have a very high $k$ value. If $k$ is one, then that grammar is called
**LL(1) grammar.** These grammars are very restrictive but very popular because the corresponding LL parsers only need to look at the next token to make their parsing decisions.

**LR(k) grammar.** This is the grammar for which right parsers (also called bottom-up parsers) [3] can be built deterministically if the parser is allowed to look $k$ input symbol (tokens) beyond its current input position.

Now we briefly describe how an $LL(k)$ grammar can be transformed into an $LL(1)$ grammar for efficient top-down parsing. A detailed discussion of this grammar transformation can be found in [3, 6].

### 2.3.2 Left recursion removal in LL(k) grammar

A top-down parser for a grammar can loop forever if there is a production rule that has $A$ as a non-terminal of the form $A ::= A \alpha$ for some string $\alpha$. This is known as left recursion. There are two types of left recursion as defined below:

(a) **Immediate left recursion**

$$A ::= A x$$

(b) **Indirect left recursion**

$$A ::= B x$$

$$B ::= A y$$

We can remove these left recursions from the production rule by using substitution, as explained in the dragon book [3].

### 2.3.3 Left factoring

Left factoring is a grammar transformation technique that is suitable for
producing \textit{LL(1) grammar}. For example, in a production rule sometimes it becomes difficult to decide which alternative production to use to expand a non-terminal. To remove this ambiguity from the grammar, we rewrite the production rule such that we delay the decision until we have enough information. We explain this ambiguity with the following example:

\[ A ::= zX \mid zY \]

In the above production rule we do not know whether to expand the non-terminal \( A \) to \( zX \) or \( zY \). The above production rule can be replaced by the following production rules to remove this ambiguity:

\[ A ::= zA' \]
\[ A' ::= X \mid Y \]

2.4 Executable UML

Executable UML is a profile of UML that allows a developer to define the behavior of a single subject matter (a system) in sufficient detail, so that it can be executed [16]. These models are then woven together by an executable UML model compiler. An RFP (request for proposal) [9] is “Work in Progress” in OMG for defining executable UML. The objective of this RFP is to enable a chain of tools that supports construction, verification, translation and execution of computationally complete models. This proposal is for the definition of a computationally complete and compact subset of UML 2.0, to be known as “Executable UML Foundation,” along with a full specification of the execution semantics of this subset. This compact subset is the selected metamodel of UML, which is as small as possible, that practically achieve computational
Completeness.

Executable UML comprises the class model, the state machines for the classes and the state procedures. These three models are expressed as UML class diagram, UML state diagram and action language. In late 2001, semantics for actions [27] were added to the UML, which provides a complete set of actions as a high level of abstraction. These defined semantics makes these three models computationally complete. Therefore a UML model specified using the above three models can be executed, verified and translated into implementation. This executable UML model is separate from any implementation and can be called platform independent model (PIM).

Model compilers are used to translate executable models into implementation using a set of decisions about the target hardware and software environment. A system can be divided into domains or subsystems. A domain can be added and can be modelled using executable UML. Each domain is independent of other domains in the system and hence can be replaced by another domain. Models of each domain are woven together by specifying a set of join points between the models. The two domains are joined together through the bridges. A bridge is defined as a layering dependency between domains. A domain makes assumptions and other domains take those assumptions as requirements. The domain keeps its autonomy using the bridges. Each domain is an aspect and each bridge is a set of join points, and these domains need to be woven together and translated to form the implementation by the model compiler. This is the same concept used by aspect-oriented programming (AOP) [51], at a different level of abstraction.

2.5 OCL (Object Constraint Language)

One of the shortcomings of UML is that it cannot be used for providing all the
relevant information about the specification of a system. This relevant information is usually provided in a natural language, which is not very precise and hence is ambiguous. To remove this ambiguity we use formal languages. Because of the mathematical complexities of formal languages, it is very difficult for a non-programmer to use them for system specification. To make UML models more precise and comprehensive, OCL has been defined by the object management group (OMG).

OCL is not a programming language but a modeling language. OCL is a pure specification language: an expression in OCL simply returns a value and cannot change anything in the model. OCL is a typed language, and its expressions are written in the context of an instance of a specific type. OCL can be used as a query language. It can also be used to specify invariants and types, to describe pre and post conditions and to specify constraints on operations. The general form of an OCL expression is shown below:

```
package <packagePath> /* package name */
context <contextualInstanceName>:<modelElement> /* expression context */
    <expressionType><expressionName>: /* expression */
        <expressionBody>
    <expressionType><expressionName>: /* expression */
        <expressionBody>
    ...
endpackage
```

We use bold to show an OCL keyword and italic to show an optional element.
2.5.1 OCL collections

*Collection* is a predefined type in OCL. There are four different *Collection* types in OCL, *Set*, *OrderedSet*, *Sequences* and *Bag*. A *Set* is a mathematical set. It does not contain duplicate elements and the elements are not ordered. An *OrderedSet* is a *Set* whose elements are ordered. A *Bag* is like a *Set* that may contain duplicate unordered elements. A *Sequence* is like a *Bag* in which the elements are ordered. These collections have an extensive set of operations for conversion, comparison, query, access, selection and iteration.

2.5.2 OCL navigation

Navigation is a process of following a link from a source object to one or more target objects. Starting from a specified object, we can navigate an association to refer to other objects and their properties. The properties defined in the OCL specifications [2] are an attribute, classifiers, association ends and a side-effect-free (query) method or operation. By default, navigation always results in a *Set* when multiplicity is greater than 1. Navigation across multiple associations always results in a *Bag* when multiplicity is greater than 1.

2.5.3 OCL standard library

The OCL standard library consists of built-in types, predefined types and operations. The built-in types are *OclAny*, *OclVoid*, *OclInvalid* and *OclMessage*. The predefined types include primitive types, *Integer*, *Real*, *String*, *Boolean*, and *Collection* types (also called OCL collections). *OclAny* is a super type for all the types except the OCL predefined *Collection* types. All these types have an extensive set of operations specified in [2]. This standard library is a mandatory part of OCL and any
implementation of OCL must include this library package.

2.6 TLA (Temporal Logic of Actions)

TLA is a high-level specification language based on set theory and first order logic. This section gives an overview of TLA with the help of an example. More details on TLA can be found in [50]. A system is formally specified using a mathematical model. TLA is a tool for specifying the mathematical model of a system, which may have some behavioral properties. These properties can be functional and logical, and represent a correct execution of the system. A system can be described as a sequence of states. A state assigns values to the variables. While specifying a system's behavior we need to define an initial value, the variable's values as a relation of time and some constants.

TLA specification helps the design process and problems can be corrected in the design phase rather than after implementation. These specifications can be communicated in a clear and concise way and provide a valuable guide to engineers who implement and test the system. Since TLA specification is a formal description of the system, tools can be applied to find errors and test the system. TLC, a TLA model checker [50], is a tool that is written for this purpose and is used in this thesis to verify the CAL model specifications (see Chapter 5).

To see how TLA is used, we first start with a description of an example of a trivial system using TLA. This example is taken from the book [50] Specifying systems: the TLA+ language and tools for hardware and software engineers by Leslie Lamport. The example is a simple hour clock whose display cycles through the values 1 through 12. We can say there are 12 states in this system. If we represent the clock hour with $hr$, then the behavior of the clock can be defined as follows:
As explained above we need to specify initial values and a relation for next state formula. We can define predicates $HC_{ini}$ and $HC_{nxt}$, where the symbol $\equiv$ means is defined equal to:

$$HC_{ini} \equiv hr \in \{1, \ldots, 12\}$$

$$HC_{nxt} \equiv hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$$

While defining the formula for $HC_{nxt}$ we have used primed ($hr' = \text{value in next state}$) and unprimed ($hr = \text{value in current state}$) variables. Such a formula is called an action. Action can be TRUE or FALSE after a state. When an action occurs, we say that action is executed. In this case, when $HC_{nxt}$ occurs we say that $HC_{nxt}$ is executed. Now we specify a single formula that asserts that the initial state satisfies $HC_{ini}$, and every other state satisfies $HC_{nxt}$. For this purpose, TLA uses a temporal logic operator $\Box$ (pronounced box). The temporal formula $\Box F$ asserts that formula $F$ is always TRUE. So we can write a single temporal formula as

$$HC \equiv HC_{ini} \land \Box HC_{nxt}$$

which is only true if the initial state satisfies $HC_{ini}$, and every other state satisfies $HC_{nxt}$. This formula satisfies all the behaviors defined in Equation 2.1. There may be a state where $hr$ value does not change i.e. $hr' = hr$. These are called stuttering steps of the clock and satisfy $HC_{nxt} \lor (hr' = hr)$. In TLA, this can be written as $\Box [HC_{nxt}]_{hr}$, and we can write the temporal formula in compact form as

$$HC \equiv HC_{ini} \land \Box [HC_{nxt}]_{hr}$$
This formula also allows the following behavior with stuttering states:

\[ [hr = 10] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow [hr = 12] \rightarrow \ldots \]

For the clock to behave properly, it should only display an integer from 1 to 12 in every state of any behavior satisfying the clock specification $HC$. $HCini$ asserts that $hr$ is an integer from 1 to 12, and $\Box HCini$ asserts that $HCini$ is always TRUE and should be true for any behavior satisfying $HC$. Therefore, we can say the temporal formula $\Box HCini$ implies $HC$ should be satisfied by every behavior. A temporal formula satisfied by every behavior is called a theorem, so $\Box HCini \Rightarrow HC$ is a theorem.

A typical TLA program consists of a module name, an extends statement, a declaration statement, operator definitions and a theorem. Figure 2.1 shows a listing of the example both in typesets and an ASCII version with the configuration file (HourClock.cfg). The version with typesets is line numbered from 1 through 6. At the top, MODULE HourClock defines a module name, HourClock. The line 1 extends the Naturals module (library of natural numbers), which is one of the built-in modules of TLA. Any built-in or user-defined module can be extended and reused. Line 2 declares a variable. Lines 3, 4 and 5 are operator definitions. The last line defines a theorem. The ASCII version of the specifications can be tested and verified by TLC. In the next Section, we describe how and why an algorithm language based on TLA is used to describe a system that can be model checked by TLC.

2.6.1 An algorithm language (pluscal) based on TLA

The small set of CAL constructs hides the complexity of its internal model, but the tool that implements CAL needs to know this complex model. Therefore, we need to
formally describe this model for the tools and also for the purpose of verification. The CAL data structure (ADAG), operations and actions can be directly specified in TLA but may need significant amount of space and time. To speed up this process and save space, in our research we found an algorithm language that is based on TLA and surprisingly named C-type Algorithmic Language (+CAL). Because of the similarity of its name with the language defined in this thesis, we call this algorithm language as pluscal. In this section we briefly give an overview of pluscal.

---

```plaintext
MODULE HourClock

1- EXTENDS Naturals
2- VARIABLE hr
3- HCini = hr ∈ (1 .. 12)
4- HCnxt = hr' = IF hr ≠ 12 THEN hr + 1 ELSE 1
5- HC = HCini ∧ C[HCnxt]_hr

6- THEOREM HC ⇒ DHCini

---
```

As described above pluscal is based on TLA. There are two alternate syntaxes for

---

Figure 2.1 Simple clock example from [50]
pluscal. One is *p-syntax* and the other is *c-syntax*. Since *c-syntax* is new, more compact, and closely resembles C# and Java, we have used *c-syntax* in this thesis to describe CAL specifications. TLA expressions can be combined with *pluscal* expressions.

Figure 2.2 (a) shows a *Euclid* algorithm specified in *pluscal* using *c-syntax*. It computes the greatest common divisor (GCD) of two numbers. A procedure has been used to implement the algorithm, which is called in the *main block*. The *main block* is enclosed in brackets. If a procedure is used then we need to extend *Sequences*, to be used during translation for maintaining stack and other variables, as *Sequences* for the procedure. *TLC* is a built-in module for basic operations like *print* etc. The procedure has two labels, *RunEuclid1* and *RunEuclid2*. The TLC translates these labels into operator definitions in TLA. Each procedure should have a return statement at the end. In the *define block*, the user can define TLA expressions like *TypeInvariant* and other user-defined operators.

The configuration file is shown in Figure 2.2 (b). It sets the value of N to 9 and declares the *INVARIANT*, as *TypeInvariant*, and the *SPECIFICATION*, as *Spec*, to be checked by the TLC. Although *Spec* defined in the *Euclid.tla* file cannot be seen, it is defined automatically by the TLC translator. Every state (operator) and call to a procedure in the *main block* is a *Spec*. For example, in Figure 2.1 *HC* at line number 6 is a *Spec*. Figure 2.2 (c) is the output when *Euclid.tla* is compiled and executed by the TLC. The TLC generates 9 distinct states, and the value of *v* in file *Euclid.tla* changes from 1 to 9. That is to say, the *GCD* procedure is executed 9 times, and we see 9 different *print* outputs.
TLC is implemented in Java and requires a Java Runtime Environment (JRE) to run. The version of JRE depends on the current implementation of the TLC. The following commands were used to run the TLC translator and model checker for the file `Euclid.tla` with JRE version 1.5:

```
java pcal.trans euclid.tla
java tlc.TLC -cleanup -difftrace euclid.tla
```
2.7 Related work in model evolution

Most software evolution techniques target the source code level. There is much less support at a higher level (models) of abstraction. The major reason for this is the lack of support tools for model evolution. In this Section we discuss current efforts at automating model evolution, starting from evolution contracts by Tom Men and Theo D'Hondt in the year 2000, to the most recent efforts. A comparative analysis is done with the approach and techniques used in this thesis. The two most recent efforts, C-SAW from Vanderbilt University and the iACMTool by L.C. Briand and Y. Labiche, are compared in Section 2.7.3. Since CAL is an extension of OCL 2.0, other OCL tools have been listed and compared on the basis of the platform they support.

2.7.1 Evolution contracts

One of the earlier efforts to automate software evolution in UML was done in 2000 [28]. Tom Mens and Theo D’Hondt proposed to extend the metamodel of UML 1.3 and based on this extension defined evolution contracts using stereotypes such as \texttt{<<add>>, \texttt{<<remove>>, \texttt{<<connect>>} and \texttt{<<disconnect>>} etc. The purpose of these contracts is to automatically detect conflicts. The main idea behind this is to define a formal contract between the provider and the modifier for the evolution. They mentioned about creating a conflict table, which describes what kind of conflicts occur in different cases of conflicts, to automate the conflict detection. There is no specific detail about the table or the kinds of conflicts. Neither the data structure for storing such a table nor how the data structure was created is mentioned in the paper. The authors claim that they implemented an evolution contract framework in PROLOG for checking evolution conflicts, but no case study or example, or reference to a case study or an example, is
given in the paper.

2.7.2 Refactoring

Recently there have been many efforts at implementing refactoring [13] at the model level, which are described below. Only the efforts relevant to the thesis are included here.

The applicability of Action Semantics (AS) [27] to UML 1.4 metamodel is reviewed in [12], and an approach is presented for using AS for meta-programming to perform model transformation. The authors used this approach to propose the implementation of refactoring in UML. Their approach combined AS with OCL to verify whether a transformation may be applied to a given context. It also illustrated the use of AS for the implementation of refactoring. They show how this approach can be used to implement refactoring, which emphasizes the importance of the use of OCL with AS. The method used in this approach divided the model transformation into two types, model manipulation and code generation. Model transformation has two parts, the selection of model elements concerned and the actions performing the transformation itself. Since there is no formal specification given in the paper for the actions, we cannot analyze this approach further. Also the approach is based on meta-programming, and is not easily used by a non-programmer.

The authors in [14] applied the concept of refactoring at a higher level of abstraction. They developed a model refactoring tool where users can specify the transformation rules. This tool is front-end of a transformation engine, constraint specification aspect weaver (C-SAW), which is a plugin within Generic Modeling Environment (GME). The tool, C-SAW and GME [20, 21] are discussed below and
compared in Section 2.7.5.

Refactoring in executable UML is discussed in [15]. The computer-aided software engineering (CASE) tool used is TAU from Telelogic. A specification template for refactoring is proposed. OCL is used to formally define the specification for refactoring. The changes are carried out based on pre and post conditions. If pre conditions are met, then the transformation is carried out. The transformation is verified using the post conditions. To make these changes a TAU model is manipulated using component object model (COM) application programming interfaces (API's) implemented in Borland Delphi's Object Pascal. Although the approach used is very practical, the methodology used and the implementation are based on Telelogic TAU, and so it is not interoperable. For a non-programmer the use of COM API's is not easy.

Because the previous efforts we have discussed were concerned with refactoring, no impact analysis was carried out before making changes in these studies. In what follows, we describe two of the recent research efforts in model evolution for impact analysis. Then we discuss two other recent research efforts on modeling languages for model transformation and see if they provide any support for model evolution in the language.

2.7.3 Model evolution and impact analysis

L.C. Briand, Y. Labiche and L. O'Sullivan [22, 23, 31] proposed an approach to perform impact analysis of UML model elements and present promising empirical results. This is the first empirical effort at automating impact analysis of UML models, but it does not address making changes to the model. The changes in two versions of the model are automatically identified and reported. The impacted elements, whether direct
or indirect, are determined using impact analysis rules defined in OCL. The distance is measured between the changed element and the impacted element, to prioritize the impact analysis strategy. If the distance is large, then the chance of impact is less likely. The distance measured between the changed element and the impacted element is defined as the number of impact rules applied to identify the impacted element. These rules are used to trace the “impact of a change” as defined in definitions 2 and 3 in [22], so we can say this approach uses a traceability analysis technique for impact analysis. A prototype tool, iACMTool, has been implemented to automate the process. Impacted model elements are detected in the form of OCL collections (as Bags) by using 97 rules [31] defined in OCL. Since an element may be impacted by several others, these OCL collections may contain several occurrences of the impacted element, which greatly affects the efficiency of the tool implemented.

In the research paper [17], impact analysis is performed using traceability analysis, which unlike dependency analysis gives a coarse evaluation of relationships between program entities. However, the authors claim that their traceability analysis technique gives detailed guidance as to what traces should be established to support impact analysis. Their trace model is based on three relationships: representation, refinement and dependency. They use the Rhapsody CASE tool, from Telelogic, and a requirement management tool, RequisitePro (also a commercial tool). Both these tools are linked using the COM API's of each tool. They call this tool environment QuaTrace, which supports requirements engineers, project planners and software maintainers. The tool is good for tracing changes in requirements and documents, neither of which needs a detailed analysis of dependencies. QuaTrace is highly dependent on other tools and is not
 interoperable, and the use of API's makes this approach difficult to implement for a non-programmer.

2.7.4 Recent languages for model transformation

Naoyasu Ubayashi and others developed the aspect-oriented modeling language AspectM (aspect for modeling) [18]. AspectM is used [19] to demonstrate the effectiveness of aspect-orientation in terms of model evolution. That is, the model can be evolved by adding aspects as concerns, like security etc. AOP [30, 51] is based on Join Point Model (JPM), which includes join points, pointcuts and advice. The authors used the same concept at the modeling level. Advices can be, add class, delete class, add operation, delete operation, add attribute and delete attribute etc. Since these aspects or concerns cross-cut many places in the model, it is very important to perform impact analysis, before adding or removing these aspects. AspectM is a model compiler that supports modeling level aspects and is defined as an Extendible Markup Language (XML) based AOP language. JPM can be described in XML or AspectM notations (diagrams). AspectM supports the MDA process and weaves UML and aspect diagrams together into PIM using aspect libraries to generate PSM. However, AspectM only supports adding aspects that are not dependent on others and performs no impact analysis. It does not support pre and post conditions or invariants, and hence there is no support for defining contracts. The authors are working on extending AspectM to add support for defining contracts. They proposed a language, contract writing language (COW) [29] that can be used for defining contracts for weaving aspects. It is implemented for AspectJ [30]. Using these contracts, a programmer can specify how a program should behave before and after weaving. The work is in progress for applying
these contracts for impact analysis. AspectM is more of a model transformation language using aspects. Since AspectM does not support model evolution analysis, and the work is in progress, we do not discuss it here any further.

C-SAW [20] from the Institute for Software Integrated Systems (ISIS), Vanderbilt University, is a transformation engine for manipulating models that shows how tools can be used to automate model evolution. A tool (an engine) is developed [14] to specify the transformation rules to act as a front end to C-SAW. This engine is called an aspect weaver. C-SAW is a plug-in for Vanderbilt University’s GME, a configurable toolset that supports domain specific modeling environments. The aspects and strategies for model weaving are based on the language ECL (embedded constraint language) [21]. It is used for describing the location and behavior of the change to be performed on the source models. Like CAL, it provides operators to change the model, addModel, addConnection, removeModel and removeConnection etc. ECL is more of an aspect oriented language like AspectM [18], for model transformation. There are two types of constructs in ECL, a strategy for defining a procedure for transformation and an aspect for defining an entry point of model transformation. Transformation rules (behavior of the change) can be specified in the form of a strategy, but it is not clear if any impact analysis is performed before transforming or making changes to the model, which is very important for maintaining software. This approach is very useful in applying multiple changes to the model at one time. ECL is implemented using GME and uses the same type system (a graph model), so unlike CAL, it is a domain specific language. A parser is implemented to generate an abstract syntax tree (AST), and an interpreter is implemented to traverse the AST and to perform transformation using modeling API's provided by
GME. ECL borrows some concepts from OCL but is not a complete extension of OCL. There are no OCL-like syntaxes for pre and post conditions for defining contracts in ECL. Only pre conditions can be applied inside strategies using a different syntax than OCL.

2.7.5 Comparison

In this Section, we compare the research that is most recent, as described in the above sections, with the approach and techniques used and the tool implemented in this thesis. Following is a comparison with the approach proposed by L.C. Briand and Y. Labiche [22]:

- The major difference is that iACMTool uses a traceability analysis technique, and CAL uses a dependency analysis technique for impact analysis.
- The approach described needs to compare two complete models to detect changes. That is, the original model and the changed model are required to complete the detection process. We use agile methods [10]. That is, a change or few changes are made and then analyzed, to keep up with the ongoing changes to the requirements. Our approach works on just one model. As the model is changing the tool is analyzing the change. It makes the analysis process practical, dynamic and compact.
- Both the approaches are formally defined and can be implemented independently or integrated into any UML tool, as is shown by their prototype tool implementations.
- The use of constraints, as rules, defined in OCL by itself is not optimal for
impact analysis. Invoking constraints each time for tracing the change of a UML model element is not efficient. For example, in [22] `select` and `equals` have been used to query the model and to match the pattern respectively. These rules generate a list of `Bags` and a counter whose number prioritizes impact analysis. The tool also generates an impact analysis report. Since CAL is a complete extension of OCL, the same rules can be used and written in CAL. In addition CAL defines an efficient data structure, ADAG that can be viewed as a dependency analysis tree. This view is helpful for software engineers and maintainers to get a full picture of the dependency tree and make early software design and maintenance decisions. It can also be interacted with to make changes.

- Distance has been used in [22] to prioritize impact analysis. The CAL data structure, ADAG, is also a transitive closure of relations, so the distance between any two model elements in the ADAG gives the same distance. In addition, we are assigning weights to each element in the ADAG so that each dependency relation can be counted according to its weight in the dependency path.

Following is a comparison of CAL with C-SAW and ECL [20, 21] from the ISIS, Vanderbilt University:

- Our research indicates that C-SAW and ECL perform model transformation without doing any impact analysis of the changes being made. The approach
used is more like software automation for changes in a GME model than software evolution.

- C-SAW is a plug-in for GME, which is a domain-specific modeling toolset. It is based on the GME graph model and uses GME API's to manipulate models. That is, C-SAW and ECL are dependent on another toolset, GME, and cannot be integrated into other tools or used independently.

- ECL is more of an aspect-oriented language like AspectM [18]. Using *aspects* and *strategies*, it can be used for making changes to the GME model. Although it is not mentioned in the paper, *strategies* can be used to search for dependent elements. The same can be achieved by using constraints in OCL. However, ECL has never been used for this purpose, and it is not possible to obtain all the direct and indirect dependencies using this approach.

- ECL automates the process of transformation using *aspects* and *strategies* as described above and needs an aspect weaver, such as C-SAW. If a change is made in one of the *aspects* then, how does that change affect the weaving with other *aspects*? How does that change affect the weaved model? *Aspects* are added assuming that they are independent of other *aspects* in the model, but can we add and delete *aspects* that are dependent on others? In reality software evolution is about changes made to the software that can either be dependent or not on other objects. There is no mention of such support in ECL or C-SAW while defining *aspects* or *strategies*. These questions may open up another research area for software evolution in aspect-oriented modeling (AOM) or in domain-specific languages (DSL).
• ECL is not a full extension of OCL. The syntax of ECL is different than OCL. Only pre conditions can be applied inside strategies on GME models. There is no verification of the changes made using ECL. Verification is only done through GME. CAL is a complete extension of OCL, and changes can also be verified using OCL constraints.

2.7.6 Impact analysis support in commercial UML tools

This Section gives a brief description of commercial tools that can be used for impact analysis of UML models. Rhapsody [32] provides COM API’s that developers can use for impact analysis. Rational in some of its products like Rational Rose [33] and XDE [34] provides API's that can be used in Java, Visual Basic or C# to manipulate a UML model. These API’s can also be used to develop a separate application or a tool, or can be integrated within the project to perform impact analysis. Other tools (see Section 3.5) have their own action languages, which can be used for model execution and transformation but not for impact analysis. Rhapsody Gateway [35] can be used for impact analysis. Gateway is only a requirements traceability tool that is a Rhapsody add-on. It uses traceability techniques to perform impact analysis. There are significant disadvantages and practical concerns connected with these tools and the options they provide which are summarized below:

• It takes much effort and skill from an average software engineer who is not a programmer to develop an impact analysis application or a tool using the API's provided by the tool vendors. If not developed properly the application
or the tool may need to be developed or changed for each project.

- Because of the limited capabilities of these API's the developer may not be able to define impact analysis rules at a higher level of abstraction.
- The tools and the options these tools provide are not automatic or portable and are highly dependent on the specific tool.
- There is a need to automate these tasks and make them flexible, interoperable and independent of any CASE tool.

Table 2.3 lists the tools that support OCL, including commercial, academic and open source, with a brief description of each tool. Only those tools are included that support OCL 2.0. The reason for listing them here is not to compare their technical features, but to compare the platform they support and where they are used (i.e. the license). Unlike the CAL parser, all the academic and open source tools are implemented in Java. Commercial tools are implemented using C++. This shows that the OCL tools that are being used in industry are still implemented in C++. Almost all the commercial CASE tools that support UML are implemented in C++ because of the graphical nature of the UML diagrams and notations. With the increasing speed of a central processing unit (CPU) and a graphics processing unit (GPU), this may change in the future.
### Table 2.3 Current OCL tools

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Description</th>
<th>Company</th>
<th>Platform</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>Octopus</td>
<td>An OCL plugin for eclipse</td>
<td>Klasse Objecten octopus.sourceforge.net</td>
<td>Java</td>
<td>Open Source</td>
</tr>
<tr>
<td>Oclarity</td>
<td>Add-in for Rational Rose to support OCL</td>
<td>EmPowerTec</td>
<td>Rational Rose (C++)</td>
<td>Commercial</td>
</tr>
<tr>
<td>Dresden OCL Toolkit</td>
<td>OCL parser</td>
<td>Dresden University of Technology</td>
<td>Java</td>
<td>Open Source</td>
</tr>
<tr>
<td>OSLO (Open Source Library</td>
<td>Based on the OCL implementation of the University of Kent</td>
<td>University of Kent oslo-project.berlios.de</td>
<td>Java</td>
<td>Open Source</td>
</tr>
<tr>
<td>Bold for Delphi</td>
<td>Add-in to Borland Delphi IDE to support OCL</td>
<td>Borland Software Corporation</td>
<td>Delphi (C++)</td>
<td>Commercial</td>
</tr>
<tr>
<td>The KeY Project</td>
<td>It is a formal software specification and verification tool. OCL is used as a basis for reasoning about the specification. It requires Borland Together for UML / OCL support.</td>
<td>University of Karlsruhe i12www.ira.uka.de/~key</td>
<td>Java</td>
<td>Academic</td>
</tr>
<tr>
<td>OCLE (Object Constraint</td>
<td>OCL tool</td>
<td>University of City-Napoca</td>
<td>Java</td>
<td>Academic</td>
</tr>
<tr>
<td>Language Environment)</td>
<td></td>
<td>lci.cs.ubbcluj.ro/ocl/index.htm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCL Compiler Version 1.5</td>
<td>This is a demo version and an industrial version is under development</td>
<td>Cybernetic Intelligence GmbH</td>
<td>Windows</td>
<td>Commercial</td>
</tr>
</tbody>
</table>
Chapter 3

CAL

Constraints with action language (CAL) is a text based, scripting language defined for the purpose of model evolution, and can be extended and used as an action language or in model compilers for incremental recompilation and retransformation of models, in model driven architecture (MDA) [11, 47]. CAL contains a small set of constructs, but is powerful enough to be used efficiently for typical software evolution management operations like impact analysis, correction, improvement and enhancement of models. CAL is an extension of object constraint language (OCL) and is not specifically defined for UML. It supports generating the Annotated Directed Acyclic Graph (ADAG) of the model for dependency analysis.

The current implementation of the prototype tool VCAL (visual CAL), which will be discussed in Chapter 6, interprets CAL statements and then uses an Extensible Markup Language Metadata Interchange (XMI) representation of Unified Modeling Language (UML) to manipulate UML Class Models. The CAL parser also to be discussed in chapter 6 is implemented separately from the tool VCAL that uses it. In CAL we have extended OCL and added 5 actions, 10 operations and 1 data type (the data structure ADAG). Since CAL is a complete extension of OCL, we do not describe OCL types, operations, library or usage. Interested readers are referred to Chapter 2 and OMG document [2] for readings on OCL.
In the present chapter, we discuss the extended CAL grammar, library and usage, and compare it with other action languages for software models. We also describe using OCL-like syntax CAL actions, operations and its data type. Within this description, the word *result* is used to refer to the value that is obtained from evaluating the operation and *iterate* is any searching technique (e.g. depth first search or breadth first search etc.) suitable for a directed acyclic graph (DAG). Post conditions are used to describe the properties of the result. At the end of this chapter, we compare CAL with other similar languages (action languages). Before describing CAL in detail, we first, list its basic characteristics:

- **Complete extension of OCL**, so constraints can be used with actions. Hence, CAL combines both declarative and imperative programming paradigms.

- **Automation for dependency analysis**.

- **Easy to learn**. Abstract the details so even a non-programmer in a business environment can use it effectively.

- **Well defined grammar** in a standard EBNF [1] form.

- **Formal specifications in TLA** [50] for better comprehension, implementation and verification.

- **CAL parser implemented in C++**, which can easily be integrated into other modeling tools.

### 3.1 CAL grammar

There are commercial and open source tools that support OCL that are listed in Table 2.3. None of the open source parsers is in C++, and the OCL commercial tools are
highly dependent on the tool vendor IDE. There is no open source OCL grammar for
LL(1) parsing, so we decided to start from scratch and write our own grammar for OCL
and extend it for CAL. We have used formal grammar to define the language for its
obvious advantages of making parsing and translation of CAL simpler. The grammar for
OCL 2.0 has been extracted from the OMG document [2]. The extracted grammar has
been extended and production rules have been added for different actions. The CAL
grammar can be defined by a 4-tuple \( G = ( T, N, P, S ) \), where

\[ \begin{align*}
T &= \text{set of terminals; } \\
N &= \text{set of non-terminals; } \\
P &= \text{set of production rules; } \\
S &= \text{an element of } N, \text{ a distinguished starting non-terminal. }
\end{align*} \]

The CAL grammar has been optimized for LL(1) parsing by removing left
recursion and using left factoring in the parser, as explained in Section 2.3 and 6.2. In the
process, a new set of production rules, such as \textit{LogicalStmt}, \textit{RelationalStmt}, \textit{AdditiveStmt},
\textit{UnaryStmt} and \textit{MultiplicativeStmt}, which are not defined in the concrete syntax of OCL
[2], has been added in the grammar for different operations.

The complete listing of the CAL grammar in EBNF [1] is given in Appendix A.

There are three basic statements in the grammar, \textit{OclStmt}, \textit{ActionStmt} and \textit{DefStmt}. A
CAL file starts with \textit{package} and ends with \textit{endpackage}. In between comes the
statements, which are of three types, \textit{attrOrAssocContext}, \textit{classifierContext} and
\textit{operationContext}. They all end with either \textit{DefStmt}, \textit{OclStmt} or \textit{ActionStmt}.

\subsection*{3.1.1 Production rules added for CAL}

There are five kinds of action statements that have been added for CAL:
createStmt, deleteStmt, addAttrStmt, createLinkStmt and deleteLinkStmt. As each name suggests, they can be used to create or delete methods or attributes, and create or delete links or associations between elements. These production rules have been added to perform model evolution (change) tasks, and they are informally defined and explained below as CAL actions. The deleteStmt deletes both operation and attr (id can be the name of an operation or an attr). Figure 3.1 shows a complete list of the rules for these statements. Action statements can be used only in an operation context. We also list in

```
operationContext ::= operationName

| 'pre' id? ':' OclStmt )+
| ( 'post' id? ':' OclStmt )+
| ( 'body' id? ':' OclStmt )+
| ( 'action' id? ':' ActionStmt )+

ActionStmt ::= createStmt

| deleteStmt

| addAttrStmt

| createLinkStmt

| deleteLinkStmt

createStmt ::= 'create' operation

| 'create' operation 'to' pathName ;

deleteStmt ::= 'delete' id

| 'delete' id 'of' pathName ;

addAttrStmt ::= 'addattr' id

| id 'to' pathName ;

createLinkStmt ::= 'link' pathName association pathName ;

deleteLinkStmt ::= 'unlink' pathName pathName ;

operation ::= visibility* id '(' parameters* ')' ;

pathName ::= id ( '::' id )* ;

visibility ::= 'public' | 'private' | 'protected' ;

id ::= letter+ allowedChar* ;

allowedChar ::= digit | '!' | ',' | '$' | '%' | '#' | '&' | '?' | '~' ;

letter ::= ['a' - 'z'] ['A' - 'Z'] ;
```

**Figure 3.1** Production rules for CAL

Figure 3.1 the operationContext where ActionStmt has been defined. Pre conditions should come before and post conditions should come after the action statements. In one operation context there can be more than one action statement. An action statement is
only valid for the operation context in which it is used and cannot be used outside that operation context.

3.2 CAL library

Like OCL, CAL is a typed language, so we have defined another collection type DagType for the CAL data structure ADAG. In addition to the OCL library defined in [2], CAL contains an additional library function Dag. This library function can be used to create the ADAG of a UML Class Model as defined below in Section 3.3.1. The methods depend and descendants, defined below, check for dependency between two elements and all the descendants of an element respectively. The production rules deleteStmt and deleteLinkStmt added for CAL check the dependency using the library function depend, which is defined in section 3.2.9, and accordingly the dependent objects are deleted or unlinked. The collection type DagType in CAL is used to define the ADAG. The complete process of using the ADAG can also be displayed visually to let the software design or maintenance engineer decide before making any changes to the model. In the next sections, we present the definitions of all the types and operations in the CAL library. These definitions use one of the ADAG operations contains, which is defined in Section 4.3.

3.2.1 DagType

DagType is a collection type, a place holder for the ADAG and a subclass of CollectionType. It can be used to create the ADAG using the following syntax. The name of DagType is Dag followed by the element's name:

\[ \text{inv: } \text{self.name} = \text{Dag}\{\text{'} \text{+ self.element.name} \text{+ } \text{'}\} \]
3.2.2 Dag

Dag is a DAG of an instance of the context (see Section 3.3.1 for the syntax and Section 3.4 for an example). The instance becomes the root vertex, and all its dependent vertices become the child vertices in the form of a DAG. DagType D is represented by a tuple of four: V the set of all the vertices, E the set of all the edges, L the set of all the labels and W the set of all the weights. We call D the annotated DAG (ADAG), which is the CAL data structure, and is formally defined as follows:

\[ D = (V, E, L, W) \]

Where \( E = \{ xy: x \in V, y \in V, xy \in E \} \)

such that

E is a Partial order (transitive and non-reflexive) on set V

\[ L = \{ (l(x) \cup l(y)) \in L : xy \in E \} \]

l(x) is the label of x and l(y) is the label of y. Read chapter 4 for an example and how to compute the label

\[ W = \{ w(x, y) \in W : xy \in E \} \]

w(x, y) is the weight of x that is based on the relation (x→y) between x and y. The rule for computing w(x) is formally defined in section 5.1.

Total weight of v ∈ V is defined as

\[ w(v) = \sum_{(a,b) \in D(v)} w(a,b) \] 3.1

where D(v) ∈ E and is a set of descendants of v
3.2.3 `isdag`

This operation returns TRUE if `self` is `DagType`; else it returns FALSE.

*Context* `Dag::isdag() : Boolean`

*post* : `result = self-->type-->elementType-->oclIsKindOf(DagType) = 0`

3.2.4 `size`

This operation computes the number of vertices in the ADAG.

*Context* `Dag::size() : Integer`

*post* : `result = self-->iterate(elem; count : Integer = 0 | count + 1)`

3.2.5 `isempty`

This operation returns TRUE if the ADAG is empty; else it returns FALSE.

*Context* `Dag::isempty() : Boolean`

*post* : `result = Dag-->size() = 0`

3.2.6 `flatten`

This operation returns the same element if the element type is not the `DagType`. If element type is a `DagType` then the result is an `OrderedSet` containing all the elements of `self`.

*Context* `Dag::flatten() : OrderedSet(T)`
post: result = if (self-->isdag())
    then
        self-->iterate(elem; s : s-->union(elem-->asOrderedSet()))
    else
        self
endif

3.2.7 includes

This operation checks if object $T$ is present in the ADAG.

Context Dag::includes(object : T) : Boolean
pre: self-->isdag() and (not self-->isempty())
post: result = self-->contains(T) = 1

3.2.8 excludes

This operation checks if object $T$ is not present in the ADAG.

Context Dag::excludes(object : T) : Boolean
pre: self-->isdag() and (not self-->isempty())
post: result = self-->contains(T) = 0

3.2.9 depend

This operation returns TRUE if an object parent is dependent on an object child.
These two objects must be elements of the ADAG. It computes the mod of two elements to check for dependency. If mod is zero, then the two elements are reachable, i.e. dependent on each other \((parent \rightarrow \text{child})\). The post condition computes the mod of labels of the object \(parent\) and the object \(child\). The ADAG is annotated (labeled) in such a way that if any two labels are divisible then the two objects are dependent and vice versa. Hence, if the mod is equal to zero, then the two objects are dependent. This improved labeling scheme for the ADAG is described in Chapter 4.

\[
\text{Context Dag::depend(object parent, object : child) : Boolean}
\]
\[
\text{pre: self--->isDagType() and self--->includes (parent)}
\]
\[
\text{ and self--->includes (child)}
\]
\[
\text{post: result = (if label(parent) > label(child)}
\]
\[
\text{ then}
\]
\[
\text{ label(parent) mod label(child)}
\]
\[
\text{ else}
\]
\[
\text{ label(child) mod label(parent)}
\]
\[
\text{ endif ) = 0}
\]

### 3.2.10 descendants

This operation returns all the *descendants* of the object \(E\) as an ordered set.

\[
\text{Context Dag::descendants(object E) : Boolean}
\]
\[
\text{pre: self--->isDagType() and self--->isempty()}
\]
post: result = self--->iterate(elem; s:self--->depend(E, s))->flatten()

3.2.11 view

This operation creates a view of the element $E$ as the root element and returns TRUE if a view is created. The view displays, either as text or graphics depending on the tool that implements the compiler, the ADAG with all its vertices and edges. A user can use this view to perform all 5 CAL actions on this interactive ADAG.

$Context\ Dag::view(object : E) : Boolean$

$pre: self--->isdag() and ~self--->isempty()$

$post: result = Dag \{ E \}--->view()$

3.3 CAL actions

CAL actions, create, delete, link, unlink and addattr, are responsible for incrementing the global variable CHANGE_RATE (global variable defined in the CAL library). CHANGE_RATE is the number of objects changed and can be used to calculate the percentage of objects that changed from one version to the next, which is called change rate [4]. Similarly create and delete increment or decrement the global variables OBJECTS_ADDED and OBJECTS_REMOVED respectively to calculate the growth rate [4], which is the difference between added and removed objects. The increment and decrement of the global variables CHANGE_RATE, OBJECTS_ADDED and OBJECTS_REMOVED are not part of the definitions. They are part of the 5 DAG operations used in the definitions. One of the actions, create, and its sub-actions, setweight and relabel,
are formally specified in TLA+ for verification in Chapter 5, so we do not define \textit{setweight} and \textit{relabel} in this section. In this section, we informally define all the CAL actions in OCL-like syntax. These definitions use CAL operations (\textit{isdag}, \textit{size}, \textit{includes}, \textit{excludes}, \textit{depend} and \textit{descendants}), which are defined above, and DAG operations (\textit{add}, \textit{delete}, \textit{setParent} and \textit{removeLink}), which are defined in Section 4.3.

3.3.1 create

This action adds an object to the CAL data structure. The ADAG operation \textit{add} labels the object as 1 (default label) and sets its default weight as 0.

\begin{verbatim}
Context Dag::create (object : T) : Boolean
pre: self--->isdag()
body: self--->add(T)
post: result = self--->includes(T)
\end{verbatim}

3.3.2 delete

This action deletes an object from the CAL data structure, relabels the ADAG (only the ancestors of the object) and resets the weight (only the ancestors of the object).

\begin{verbatim}
Context Dag::delete (object : T) : Boolean
pre: self--->isdag() and self--->includes(T)
body: if (self.descendants(T)--->size() = 0)
then
\end{verbatim}
self->setweight(null, T)
and self->relabel(T)
and self->delete(T)
else
self
endif
post: result = self->excludes(T)

3.3.3 link

This action links (creates a relationship) two elements \((P = parent\) and \(C = child\)) in the CAL data structure, labels \(P\) object and sets \(P\)'s weight according to the relationship \(A\) (association).

Context Dag::link (object : A, object : P, object : C) : Boolean
pre: self->isdag() and self.includes(P) and self.includes(C)
body: self->setParent (P, C) and self->setweight(A, P)
and self->relabel(P)
post: result = self->depend(P, C)

3.3.4 unlink

This action unlinks (deletes the relationship) two elements \((P = parent\) and \(C = child\)) in the CAL data structure, labels the \(P\) object and sets \(P\)'s weight.
Context Dag::unlink (object : P, object : C) : Boolean

pre: self-->isdag() and self.includes(P) and self.includes(C)

body: if self-->depend(P, C)

    self-->removeLink (P, C)

    and self-->setweight(null, P)

    and self-->relabel(P)

else

    self

endif

post: result = ~self-->depend(P, C)

3.3.5 addattr

This action adds an attribute A to object C in the CAL data structure.

Context Dag::addattr(object : A, object : C) : Boolean

pre: self-->isdag() and self.includes(C)

post: result = self-->setAttribute(A, C)

3.4 Usage

This section describes the usage of all the CAL actions. Figure 3.2 shows a sample CAL file. The model used is of the CAL parser designed and implemented as part of this thesis. The CAL parser’s UML Class Diagram is shown in Appendix D and its
XMI file, with details of classes and their relationships, is shown in Appendix F. The same model is used as an example in the case study described in Section 6.4.

--- Sample file created on January 14, 2007 for testing VCAL
package sample

context Model::addClass ():
1. pre: Dag { Model }
2. action: create CodeGen()
3. action: link AST Dependency CodeGen
4. action: link Error Dependency CodeGen
5. action: link LG Dependency CodeGen
6. post: AST->descendants()->includes(CodeGen)
7. post: Error->descendants()->includes(CodeGen)
8. post: LG->descendants()->includes(CodeGen)

context Model::removeLinks ():
9. pre: Dag { Model }
10. pre: CodeGen->descendants()->size() < 1
11. action: unlink AST CodeGen
12. action: unlink Error CodeGen
13. action: unlink LG CodeGen
14. post: not CodeGen->depend(AST)
15. post: not CodeGen->depend(Error)
16. post: not CodeGen->depend(LG)

context Model::deleteClass ():
17. pre: Dag { Model }->isempty() > 0
18. pre: CodeGen->descendants()->size() < 1
19. action: delete CodeGen
20. post: Model->excludes(CodeGen)

context Model::flatten():
21. body: Dag { Model }->flatten()

context Model::viewDag():
22. body: Dag { Model }->view()

endpackage

Figure 3.2 Sample CAL file

This sample CAL file shows how to use all the 5 CAL actions defined in Section 3.3. There are 5 operation contexts listed in this CAL file. The first operation context adds a class to the model. First we create the ADAG of the model in line 1 (using a pre
condition, i.e. the type is a DagType). In line 2, we create (add) a classCodeGen in the model. Links are created from classes AST, Error, LG to CodeGen class in lines 3, 4 and 5 respectively. All these links are type Dependency. Weights of these classes are set by the CAL interpreter according to this link type. In lines 6, 7 and 8, using post conditions we check if classCodeGen is included in the descendants of AST, Error and LG classes respectively.

In the next operation context, removeLinks, we want to remove the linksAST-->CodeGen, Error-->CodeGen and LG-->CodeGen, so that the classCodeGen can be removed later in the following operation context. Lines 11, 12 and 13 remove these links. In lines 14, 15 and 16, we make sure that there is no link between classCodeGen, and AST, Error and LG classes.

The operation context deleteClass deletes the classCodeGen. In the pre conditions in lines 17 and 18, we check if the ADAG is not empty and there is no descendant of classCodeGen. The class is deleted in line 19. In post condition at line 20, we check if classCodeGen is not in the model.

The next operation context is simply to show how to use library operation flatten of the ADAG. It returns a tuple of 3 containing the set of all the vertices, two sets of weight and label of each vertex in the ADAG.

The last operation context is the most efficient way of using CAL actions. It creates an interactive view of the ADAG with all its vertices and edges. All 5 CAL actions can then be performed from this interactive view.

3.5 CAL and other action languages

This section discusses and compares action languages and their support for impact
analysis. It is outside the scope of this thesis to cover all such action languages. Only those that are related to software models at a higher level of abstraction and specifically used for model driven software engineering (MDSE) are discussed. We have not included the action languages discussed in [16], as they have more or less the same characteristics as the action languages discussed in this thesis, and as they are also not currently used in any industrial or academic settings. Table 3.1 gives a comparison of CAL with other action languages that are mentioned in this thesis and are described below.

**OAL** (object action language) [24] is an action language that is fully supported by the Nucleus BridgePoint UML Development Suite from Mentor Graphics (www.mentor.com), a company specializing in electronic design automation (EDA) tools. OAL is based on action semantics (AS) [27] specifications of UML 1.5, which is a separate specification document and was not part of UML 1.5. Starting from UML 2.0, the AS specifications are part of the UML 2.0 superstructure [2]. OAL was originally designed for the models used with the BridgePoint UML Development Suite. OAL is being used for the execution and transformation of UML models and is implemented in Mentor Graphic's BridgePoint UML Suite.

**ASL** (action specification language) [25] is an action language also based on AS, which is now part of the UML 2.0 superstructure as described above. It is designed and implemented to work with iUML the executable UML modeling CASE tool from Kennedy Carter Limited (www.kc.com). In terms of functionalities it is more powerful than OAL. It supports complex data structures and creation of new objects (see Table
PAL (platform independent action language) [26] is an action language that also supports the executable UML and is part of the PathFinder's (www.pathfindermda.com) PathMate transformation engine which is part of their PathMate Model Automation and Transformation Environment. PAL is also based on AS and supports complex data structures. The syntax of PAL is more like C++.

Now we make a comparative analysis of these three action languages with CAL in Table 3.1. Table 3.1 does not give an extensive comparison of these languages, which is not the purpose of this thesis. The rationale for the comparison of these languages with CAL is that they are based on actions and they are capable of manipulating (changing) a UML model. We would like to see if they provide any support for impact analysis or for using constraints. We would also like to explore the potential and advantages of CAL for use as an action language. The main purpose of these action languages is to make UML models executable. Unlike OCL they can be used to make changes, but these changes do not affect the model, because, except for ASL, they can only instantiate an object but cannot create a new object in the model. They can also be used to navigate the model, just like OCL.

These action languages are designed with the design part of the software life cycle in mind, and hence they lack software maintenance capabilities (impact analysis, propagation of changes, refactoring etc.). Code can be generated in different languages like Java or C++ from UML models, or the models can be directly executed on a specific
platform. They all are similar to OCL, but their syntax is quite different from OCL. CAL completely supports OCL concrete syntax (an OMG standard). All of these action languages support AS and AS does not have any concrete syntax, so each language has its own syntax.

These action languages are more powerful than CAL in manipulating the models; however, CAL is an academic effort and its purpose is to prove a concept, which is very new to the models and which none of them supports. Moreover, CAL can be extended to accommodate all that is offered by these languages. The main question that comes to mind here is why we need another language with a different syntax than OCL to manipulate models. Why not extend OCL, add actions like CAL and you obtain the same powerful language at a higher level of abstraction to manipulate models? They use AS but are not based on any industrial standard, so they are not interoperable between different tools. It is better and more advantageous to learn a language that is an extension of OCL, which is an industrial standard, that can support the same functionalities as these action languages.

From Table 3.1 it is clear that these languages are quite different from OCL. One of the purposes of defining a high level modeling language (UML, OCL) is that it can be used by programmers and non-programmers because it provides the same understanding of the design. The three action languages mentioned above are meant for skilled programmers. For example, the syntax for defining a structure in these languages is like a high level language (PAL's syntax is more like C++). In CAL it is more like OCL, and even simpler than OCL for a complex task of impact analysis, therefore using CAL, both programmers and non-programmers can understand and discuss the model with each
<table>
<thead>
<tr>
<th>Feature</th>
<th>CAL</th>
<th>OAL</th>
<th>ASL</th>
<th>PAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Supported</td>
<td>OCL 2.0 and based on Action Semantics of UML</td>
<td>Based on Action Semantics of UML</td>
<td>Based on Action Semantics of UML</td>
<td>Based on Action Semantics of UML</td>
</tr>
<tr>
<td>Executable UML</td>
<td>1 Partially supported</td>
<td>Supported</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Impact Analysis</td>
<td>Supported</td>
<td>Not supported</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>Extendable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Control Statements</td>
<td>If-then-else-endif (Same as OCL)</td>
<td>if-elif-else-end if</td>
<td>If-then-else-endif and switch-case-endswitch</td>
<td>if-else if-else</td>
</tr>
<tr>
<td>Loops</td>
<td>iterate</td>
<td>for-each-end for and while-end while</td>
<td>for-do-endfor and loop-endloop</td>
<td>for-each-where and while</td>
</tr>
<tr>
<td>Data Structure Supported</td>
<td>All OCL2 Collections and the 3ADAG (for impact analysis)</td>
<td>Not supported</td>
<td>6 Set with hierarchical data support</td>
<td>Group (ordered set of items) and GroupItter (iterator over Group)</td>
</tr>
<tr>
<td>Navigation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Communication</td>
<td>Messages (OCL)</td>
<td>Events</td>
<td>Signals</td>
<td>Events</td>
</tr>
<tr>
<td>Operators</td>
<td>Same as OCL</td>
<td>Almost same as OCL</td>
<td>Almost same as OCL</td>
<td>Almost same as OCL</td>
</tr>
<tr>
<td>Actions</td>
<td>Objects can be created and added but cannot be instantiated</td>
<td>Objects cannot be created or added but can be instantiated</td>
<td>Objects can be created, added and can also be instantiated</td>
<td>Objects cannot be created or added but can be instantiated</td>
</tr>
<tr>
<td>Timers</td>
<td>Not supported</td>
<td>4 External timer and 5 internal timer supported</td>
<td>Relative and absolute timer</td>
<td>Not supported</td>
</tr>
<tr>
<td>Tool support</td>
<td>An interoperable prototype tool has been implemented as part of the research</td>
<td>BridgePoint UML Suite ® by Mentor Graphics</td>
<td>CASE tool iUML ® by Kennedy Carter Limited</td>
<td>PathMate Transformation Engine ® by PathFinder Solutions</td>
</tr>
</tbody>
</table>

1 Executable UML as a model is supported for manipulation, but the generation of code or executable model / platform is not supported
2 Example of defining one (the Set) of the collections in OCL: Set { 2, 4, 1, 5, 7, 13, 11, 17 }
3 Read chapter 4 for more details
4 Time external to the application. For example October 14, 2006.
5 The internal system clock that measures the time in “ticks” for supporting real-time systems.
6 Syntax for defining the set in ASL

```plaintext
define structure <structure type name>
    <member name> <member type>
    <member name> <member type>
enddefine
```
other, which greatly reduces the time and cost of completing any software system or software project.
Chapter 4

DAG for Dependency Analysis

There are various applications for a directed acyclic graph (DAG) in computer science and software engineering. They are used in compilers to generate a parse tree with parent and child relations and to schedule instructions. They are used to represent networks and hierarchical systems of folders in an operating system, and relationships between classes and subsumption hierarchies of data available on the semantic web, such as Netscape Open Directory (ODP). In object oriented graphics the scene graphs are stored as a DAG. Whenever an operation is applied (in the scene graphs) to a parent node or nodes, it automatically propagates its effect to all the children. To date, ODP has indexed over 4 million sites (source: http://directory.mozilla.org) and is being used by large search engines like Netscape, AOL and Google.

Because of the number of important applications of a DAG, a great amount of effort [36, 38, 39, 40, 42, 43] has been put into labeling these graphs to improve typical operations on a DAG. In this chapter we show how a labeling scheme, DAG–Lite [40], has been used and improved in size (i.e., the number of digits in a label and the size of all the labels), and a dependency relationship weight (DRW) has been added to the DAG. We use this weight (DRW) for computing the dependency or relationship weight of each vertex of the DAG with its descendants. We call this an annotated DAG (ADAG), and it is used as the CAL (constraints with action language) data structure. Dependency
analysis involves examining detailed dependency relationships among program entities [7] and is one of the methods used in impact analysis. Each vertex in the ADAG is an entity and an edge represents the dependency relationship between two entities (vertices). The improved labeling scheme and the weight are used to effectively display, visually, this dependency analysis. The assignment and computation of the weight of each node are described in Section 5.2.2.3 and are formally defined as Rule 1.5 in Chapter 5 and Equation 3.1 in Chapter 3 respectively. In this chapter, we describe the improved labeling scheme using mathematical equations and present efficient algorithms for labeling and performing typical operations for dependency analysis on the ADAG.

A prototype tool VCAL (visual CAL) has been implemented to display, visually, the ADAG. VCAL and its implementation will be described in Chapter 6. All the ADAG operations are defined in OCL syntax in Chapter 3. The ADAG operations that are defined and described as algorithms in this chapter are `relabel(node)`, `label(node)`, `descendants(node)`, `contains(node)`, `depend(node1, node2)`, `setParent(parent, child)`, and `removeLink(parent, child)`. The important characteristics that are required in a labeling scheme, and operations and queries on the ADAG that are required for graphical representation of dependency analysis are summarized as follows:

- **Size** of the label. It should be as compact as possible.
- **Time** it takes for the queries to complete for effective graphical representation of dependency analysis.
- **Updates** should require minimal relabeling.
- **Reachability**: Checks if one of the vertices of the ADAG is reachable from other vertices, i.e. if a path exists between two vertices. For any two vertices $x$
and \( y \) in the graph we define it as a directed path from \( x \) to \( y \) and represent it as follows:

\[ x \rightarrow y. \]

For example in Figure 4.1 we can write \( P \rightarrow F \) is true.

- **Descendants**: Find all the *descendants* of a vertex, i.e. which of other vertices depend on the vertex. If \( V \) is a set containing all the vertices of the ADAG, then \( D(x) \) is a subset of \( V \) containing all the dependent vertices (*descendants*) of any vertex \( x \) in the ADAG. For example in Figure 4.1 we can write \( D(L) = \{ H, D, E \} \).

Vertices \( A, B, C, D, E \) and \( M \) in Figure 4.1 are leaves and have no descendants.

For the purpose of better comprehension and description of the algorithms in this chapter we assume that the ADAG is stored in a hash table (that is how the ADAG has been implemented in VCAL). We call this hash table \( DAG_{HT} \). We also assume that \( DAG_{HT} \) uses a *hash(node)* function to compute the hash of the node (node and vertex are same) for storing the node in the hash table. The global variable *size* is used to keep and set the size of the ADAG. The actual data structure for a node (a Java Class) in VCAL is defined as follows, and we use the same data structure in describing the algorithms in this chapter:

```java
class Node {
    public Object value; /* Special prime number for each node. By default it is one. */
    public int prime; /* E.g. in Figure 4.3 node P's prime = 1 and node L's prime = 17 */
    public int weight;
    public BigInteger label; /* Infinite length integer for label */
    public List parents; /* List of immediate parents */
    public List children; /* List of immediate children */
}
```
4.1 DAG Labeling

A directed graph or digraph consists of a set of vertices and a collection of ordered pairs of distinct edges. If the ordered pair \((x, y)\) is an edge \(e\) then we say that the edge \(e\) is directed from \(x\) to \(y\). If there are no directed cycles (i.e. for any vertex \(x\) there is no directed path starting and ending on \(x\)) in a digraph, then it is called a DAG. Figure 4.1 shows a sample ADAG that we use to explain the scheme for labeling the graph in this chapter. We formally define an ADAG \(G\) (the same definition is given for \(D\) in Section 3.2.2) as follows, where \(V\) is the set of all the vertices, \(E\) the set of all the edges, \(L\) the set of all the labels and \(W\) the set of all the weights:

\[
G = (V, E, L, W)
\]

where

\[
E = (x, y) = \{ xy : x \in V, y \in V, xy \in E \}
\]

such that \(E\) is a Partial order (transitive and non-reflexive) on set \(V\)

\[
L = \{ (l(x) \cup l(y)) : xy \in E \}
\]

\(l(x)\) is the label of \(x\) and \(l(y)\) is the label of \(y\)

\[
W = \{ w(x, y) : xy \in E \}
\]

\(w(x, y)\) is the weight of \(x\) that is based on the relation \(x \rightarrow y\)

The indegree of a vertex is defined as the number of edges adjacent to that vertex, and outdegree is defined as the number of edges adjacent from a vertex.

Ideally, for any two vertices we should be able to determine reachability between them by comparing only their labels. For the graph in Figure 4.1 with 17 vertices, how can we determine

\[N \rightarrow D\ or\ N \rightarrow A?\]
By inspecting the DAG and tracing the paths, we can determine that the first statement is TRUE and the second statement is FALSE. However, for a graph of more than 100 vertices it becomes, if not impossible, difficult to visually determine reachability between two distant vertices. First, it is difficult to visualize such a graph and if we can, then it takes more time and work to find the reachability and descendants of all the vertices, which is required for dependency analysis. If we compute reachability ideally as defined above, then we can achieve an efficient automation and graphical representation of the DAG for dependency analysis. No matter how complex or deep a
DAG is, the scheme presented in this chapter labels the DAG in such a way that a
graphical representation tool can compare the vertices for dependency (i.e. reachability
and descendants) by simply clicking (one-click) on the vertices, as is done in VCAL.
Using this labeling scheme, VCAL applies basic filters to the ADAG and achieves one-
click dependency analysis. The next section presents and describes this scheme.

4.1.1 Labeling scheme

We have used one of the properties of prime numbers, uniqueness of prime
factorization, for labeling the ADAG, which can be defined as any integer $n \in \mathbb{N}$ can be
factorized into unique prime numbers, and we can write

$$n = \prod_{i=1}^{f} p_i,$$

where $p$ is a prime number and $f \geq 1$. Hence, an integer $n$ can be divided by any of its
factors, and we can write

$$n \mod p_i = 0.$$

Using this property of prime numbers, we label all the vertices such that simply
by knowing the labels of two vertices we can compute the reachability between the two
vertices. Descendants of any vertex can also be computed by either taking mod or using
all the factors of the vertex's label. These two approaches are discussed below. We also
show how our approach is different and better in size than the approach used in DAG-
Lite [40].

The equation used for labeling a DAG is shown in Figure 4.2. In Equation 4.1,
$L(v)$ is only multiplied with $np$ when its label equals any one of its children's label. This
greatly improves the size of the label as compared to the label size of DAG-Lite in [40]
with an average complexity DAG.

Let $a, b \in \mathbb{N}$ and $LCM[a, b]$ as the least common multiple of $a$ and $b$, the set of children of $v \in V$, $C(v) = \{v_1, v_2, v_3, ..., v_n\}$, where $1 \leq n \leq outdegree(v)$ and $np$ is the next prime from an ordered list of primes starting with 2. Then we define two equations for labeling the DAG as follows:

\[
\begin{align*}
label(v) &= \begin{cases} 
1 & \text{outdegree}(v) = 0 \\
L(v) \times np & \text{for } 1 \leq n \leq outdegree(v) \\
L(v) & \text{else}
\end{cases} 
\end{align*}
\]

4.1

\[
L(v) = LCM[\text{label}(v_1), \text{label}(v_2), ..., \text{label}(v_n)]
\]

4.2

**Figure 4.2** Equations for computing the label of a vertex in the DAG

The complete labeled DAG using the scheme in Figure 4.2 is shown in Figure 4.3. Algorithms for labeling the ADAG are listed in Figure 4.4. All the leaves, $A, B, C, D, E$ and $M$, are labeled as 1. The prime numbers used are underlined. We can find the reachability between node $Q$ and node $I$, by computing the mod of the label of node $Q$ with the label of node $I$ as follows:

\[
l(Q) \mod l(I) = 1385670 \mod 7 = 1.
\]

Therefore, node $I$ is not dependent on node $Q$.

We can find all the descendants of a vertex by computing the mod of each vertex's label with the label of the vertex. If the mod is 0 then a descendant of the vertex is found and vice versa. For vertex $P$ in Figure 4.3 we find the following set, $D(P)$, of descendants of $P$: 

Since leaves in the DAG are labeled as 1, we do not use their label for finding either the reachability or the descendants. Instead, we use the label of their immediate parents (see Figures 4.5 and 4.6).

### 4.1.2 Size of the label

Since we are using prime numbers from an ordered list of primes starting from 2, we can calculate the maximum number $M$ that can be used for a label in the DAG using
the following equation:

\[ M = \prod_{i=2}^{m} p_i \] 4.3

where \( m \) is the maximum prime number used and \( p \) is the prime factor of \( m \)

The size of \( M \) depends on \( m \). For a DAG with \( n \) vertices, \( m \) can be calculated for worst and average cases using Equation 4.1 as follows:

**Worst case** \( m = n – 1 \): When a DAG is a straight line, then \( \text{outdegree}(v) > 0 \), and initially the label of each vertex is equal to its child's label so each vertex is multiplied by \( np \).

**Average case** \( m = n / 2 \): An example is a DAG with average complexity. In Figure 4.3 the underlined primes are the only primes used.

Not every vertex in the DAG is multiplied by an exclusive prime number. In Equation (1), \( L(v) \) is only multiplied by \( np \), when it equals one of the labels of children of \( v \) and hence reduces the size of \( m \) in Equation (3). This significantly reduces the number of digits in the label as compared to the similar scheme used in DAG–Lite, which assigns an exclusive prime number that is multiplied by the label of all the vertex's parents, as defined in Equation (1) in [22]. We have also improved the space for storing all the labels of the node in a DAG by assigning 1 as the label of leave vertices as compared to the scheme, DAG-Lite, which assigns an exclusive prime number to a leave, which is multiplied by the label of all the leave's parents.

**4.2 Labeling and dependency analysis algorithms**

The vertices can be labeled as the graph is built or with an already built graph. Based on the scheme described above we present here algorithms to relabel and compute
the label of any node in the DAG, and to find reachability between two nodes and the descendants of any node in the DAG. Here we refer to a node as a vertex of a graph.

Algorithms for these operations in C-like syntax are listed in Figures 4.4, 4.5 and 4.6. We also compute time complexity for 2 of these algorithms.
/**
 * Algorithm for finding descendants of the node in the DAG.
 * @param node: Node whose descendants are to be found.
 * @return A set of descendant nodes of the Node node
 */
Set descendants (node, dagNodes[]) {
    Set D[];
    dagNodes[] = DAG_HT.getArray ();  /* Get all the nodes as an array */
    /* Iterate the list of array of nodes and check for descendants.
    * There can be 3 options:
    * 1. label of the node in the array is 1
    *    check the label of all the parents, again 3 options:
    *    1. both labels are equal (node in the array is the descendant)
    *    options 2 and 3 are the same, as explained below
    *    2. node label is > the label of the node in the array
    *    compute MOD and if '0' store it in the Set D
    *    3. node label is <= label of the node in the array
    *    node is itself, or node in the array is not the descendant
    */
    for (int n = 0; n < dagNodes.length; n++)
    {
        if (dagNodes[n].label == 1)
        {
            parents[] = getParents (dagNodes[n]);
            for (int p = 0; p < dagNodes.length; p++)
            {
                if (node.label == dagNodes[n].label)
                {
                    D.add (dagNodes[n]);
                    break;
                } else if (node.label > dagNodes[n].label)
                {
                    if (MOD (node.label, dagNodes[n].label) == 0)
                    {
                        D.add (dagNodes[n]);
                        break;
                    }
                }
            }
        } else if (node.label > dagNodes[n].label)
        {  
            if (MOD (node.label, dagNodes[n].label) == 0)
            {
                D.add (dagNodes[n]);
            }
        }
    }
    return D;
}

Figure 4.5 Algorithm to find descendants of a node in the DAG

4.2.1 Time complexity

In this section, we compute the time complexity of the algorithms to find the
descendants of a node (see Figure 4.5) and find the reachability between two nodes (see Figure 4.6).

```java
/**
 * checkDependency: A procedure called by this procedure. It Uses
 *                  mod to check for dependency between two nodes.
 * @param node1: First node.
 * @param node2: Second node.
 * @return boolean: True if node1 and node2 are dependent else false.
 */
boolean depend (node1, node2)
{
    boolean result = false;
    if (node1.label != 1 &&  node2.label != 1)
        result = checkDependency (node1, node2);
    else if (node1.label == 1)
    {
        /* Since label is 1, check all the parent's label */
        parents[] = getParents (node1);
        for (int p = 0; p <  parents.length; p++)
        {
            /* If dependent then break */
            if (parents[p] == node2 || result = checkDependency
                (parents[p], node2))
                break;
        }
    }
    else if (node2.label == 1)
    {
        /* Since label is 1, check all the parent's label */
        parents[] = getParents (node2);
        for (int p = 0; p <  parents.length; p++)
        {
            /* If dependent then break */
            if (parents[p] == node1 || result = checkDependency
                (parents[p], node1))
                break;
        }
    }
    return result;
}
```

**Figure 4.6** Algorithm to find reachability of two nodes in the DAG

descendants: The DAG is stored in a simple list whose length is \( n \), where \( n \) is the number of vertices in the DAG. The *for loop* to find descendants in Figure 4.5 runs for the length of the list, so the total number of operations is equal to the length of the list, which is the number of vertices in the DAG. Hence, the time complexity for this
operation is $O(n)$, both in average and worst cases, where $n$ is the number of vertices.

**depend:** There is no loop in the algorithm listed in Figure 4.6 ($MOD$ is a mathematical operation without any loop), so the time complexity for this operation is $O(1)$ in an average case and $O(n)$ in a worst case, when the label of a node is 1 and the node's immediate parents is equal to all the nodes in the DAG, where $n$ is the number of vertices.

### 4.3 Updates

To add or delete a node, only the ancestors are relabeled, as shown in Figures 4.10 and 4.11. When node $R$ is added in Figure 4.10, only node $Q$ is relabeled, which in this

```java
/**
 * Algorithm for removing the link and is the opposite of setParent.
 * @param parent: Parent node.
 * @param child: Child node.
 */
void removeLink (parent, child)
{
    /* Remove parent from the list of parents of child */
    child.parents.remove (parent);
    /* Remove child from the list of children of parent */
    parent.children.remove (child);
}

/**
 * Algorithm for checking if node is present in the DAG.
 * @param node: Node to be checked.
 * @return True if node is present in the DAG else returns false.
 */
boolean contains (node)
{
    /* Computes the hash of node to get the location where the node
     * is stored. Then check that location for the presence of node.
     */
    if (node = DAG_HT[hash(node)])
        return TRUE;
    else
        return FALSE;
}
```

**Figure 4.7** Algorithms to remove the link of a node from the DAG, and check if a node is present or not in the DAG.
case does not change the label of the node $Q$. When the node $K$ is removed in Figure 4.11, only nodes $O$, $R$ and $Q$ are relabeled. Algorithms for these update operations are listed in Figure 4.9. We also present algorithms for three other operations ($setParent$, $removeLink$ and $contains$ – see Figures 4.7 and 4.8) for the DAG, which are used by the CAL actions (see Section 3.2).

```java
/**
 * Algorithm for setting the parent of a node. It creates a link parent --> child.
 * @param parent: Parent node.
 * @param child: Child node.
 */
boolean setParent (parent, child)
{
    child.parents.add (parent); /* Add parent in list of parents of child */
    parent.children.add (child); /* Add child in list of children of parent */
    /**
     * Check for cycle, if cycle then remove the link (parent and child).
     */
    if (checkCycle())
    {
        child.parents.remove (parent);
        parent.children.remove (child);
    }
}
```

**Figure 4.8** Algorithm to set the parent of a node in the DAG

### 4.4 Related work

An earlier scheme [36] by R. Khatib and N. Santoro labeled an acyclic graph using a scheme based on a minimum-distance spanning tree using post-order traversal, for optimizing routing in communication networks. There are three popular labeling schemes for graphs discussed in [38]. The first one is the **bit-vector scheme** in which, as the name suggests, a node is represented by a vector of bits. A 1 bit at some position identifies the node in a **lattice**. Each node inherits the bits identifying its ancestors or descendants in a top-down or bottom-up encoding. The second scheme is the **prefix**
scheme, which encodes the parent directly in the label of the node as a prefix using any tree-traversing technique. The third one is the interval scheme in which the node of the

```java
/**
 * Algorithm for adding a node to the DAG.
 * @param value: Value of the node to be added.
 */
void add (value)
{
    Node node = new Node ();       /* Initialize the data structure Node */
    node.value = value;
    node.prime = 1;
    node.weight = 1;
    node.label = 1;
    node.parents = new Parents();  /* Initialize list of parents */
    node.children = new Children(); /* Initialize list of children */
    DAG_HT.add (node);             /* Add node to the hash Table */
    size = size + 1;
}

/**
 * Algorithm for deleting a node from the DAG.
 * @param value: Value of the node to be deleted.
 */
void delete (value)
{
    Node node = new Node ();
    /**
     * Iterate the list of parents. Remove this node from each
     * of it's parent's list of children. Relabel all the ancestors.
     */
    parents[] = getParents (node);
    for (int p = 0; p < parents.length; p++)
    {
        parents[p].children.remove(node);
        relabel (parents[p]);
    }
    /**
     * Iterate the list of parents. Remove this node from each
     * of it's child's list of parents.
     */
    children[] = getChildren (node);
    for (int c = 0; c < children.length; c++)
    {
        children[c].parent.remove(node);
    }
    DAG_HT.remove (node);          /* Remove node from the hash Table */
    size = size - 1;
}
```

**Figure 4.9** Algorithms to add and delete a node in the DAG

label is given by an interval, such that it is contained in its parent's label. The last two
schemes discussed above are based on spanning trees.

A recent scheme that is not based on spanning trees is 2-hop labels [39]. This scheme uses hops (distance between vertices) to label the vertices. The label gives the distance between vertices. This distance can be used for dependency analysis and can be mapped to the dependency between two or more than two vertices. In future research, this distance could be combined with the weights used in this thesis for dependency analysis.

![Labeled DAG with node R added](image_url)

**Figure 4.10** Labeled DAG with node R added
Recent labeling schemes [42, 43] have proposed and applied labeling schemes for tree structured XML data. In [42] a DAG labeling scheme is applied to XML data. Before applying this scheme the XML data is modelled as a directed cyclic graph and then reduced to a DAG. After labeling the DAG using the interval scheme described above, is again converted back to a directed cyclic graph. The scheme proposed in [43] is an improvement on the prefix scheme described above. A detailed survey of labeling schemes can be found in [37].

Figure 4.11 Labeled DAG with node K removed
Currently prime numbers are, being used for labeling an XML tree [41] and a DAG [40]. This improves operations like reachability and descendants. In this thesis we have used and improved in size the labeling scheme DAG-Lite [40] for labeling a DAG.
Chapter 5

CAL Implementation

Constraints with Action Language (CAL) allows a user to perform different queries and tasks on models without knowledge of the complexity of the rule execution engine that processes and translates these models, so that these queries and tasks can be interpreted and executed by the CAL interpreter. However, a tool that implements CAL for UML models needs to parse and process these rules. In this chapter, we formally specify and explain the rules for translating a UML Class Model to the CAL data structure (ADAG) and the execution of CAL operations and actions (CAL model). The detailed implementation of the tool that implements the rule execution engine will be explained in Chapter 6. There are two kinds of rules that this engine processes: one for translating UML Class Model to the CAL data structure (ADAG) and the other for executing CAL operations and actions. CAL operations and actions are specified using TLA+ [50] for verification.

5.1 Rules for translating a UML Class Model to the CAL data structure

Before specifying these rules we first formally define a graph. A simple graph is represented by a tuple of two, V the set of all the vertices, and E the set of all the edges. The graph G is defined as follows:

\[ G = (V, E) \]

\[ E = (x, y) = \{ xy: x \in V, y \in V, xy \in E \} \]
We extract, using pattern matching techniques, sets of all the following dependency relationships between UML Class Model elements, as defined in UML 2.0 [5] and explained in Section 2.2:

a. Dependency
b. Composition
c. Generalization
d. Realization

Let \( m \) be a model element, then

**R 1.1:** \( V_m \{ (m \in V) \land \neg (\neg m \in V) \} \)

If \( x, y \) and \( z \) are model elements, and \( R_D \in (Dependency \lor Composition \lor Generalization \lor Realization) \), then

**R 1.2:** \( V_{x,y} \{ (xy \in E) \rightarrow ((x \ R_D y) \lor (y \ R_D x)) \} \)

**R 1.3:** \( V_{x,y} \{ (x \ R_D y) \rightarrow \neg (y \ R_D x) \} \)

**R 1.4:** \( V_{x,y,z} \{ ((x \ R_D y) \in (y \ R_D z)) \rightarrow (x \ R_D z) \} \)

We introduce and define another tuple \( W \) as the set of self-weight of all the vertices of the graph as follows:

\( W = \{ w(x) \in W : x \in V \} \)

**R 1.5:** \( V_{x,y} \{ (x \ R_D y) \land ((R_D \in (Dependency \lor Realization)) \rightarrow (w(x) = 1) \lor (R_D \in (Generalization \lor Composition)) \rightarrow (w(x) = 2)) \} \)

According to rule **R 1.1** only the model elements become vertices of the graph. Rule **R 1.2** states that if there is a relationship between two model elements \( x \) and \( y \) then they become an edge in the graph. Rules **R 1.3** and **R 1.4** describe the relationship \( R_D \) as
non-reflexive and transitive respectively, so by definition this directed graph is a DAG. Rule \( R \, 1.5 \) assigns a self-weight of 2 if the relationship \( R_D \) between two model elements, a supplier \( x \) and a client \( y \), is a generalization or composition relationship, and a self-weight of 1 if the relationship \( R_D \) between two model elements, a supplier \( x \) and a client \( y \), is a dependency or realization relationship. This assignment of self-weight to the model elements is due to the couplings of different relationships between the UML Class Model elements as described in UML 2.0 and shown in Table 5.1. The relationships are listed in order of coupling from high to low. The characteristics in Table 5.1 show that these relationships are transitive and non-reflexive. This further augments Rules \( R \, 1.3 \) and \( R \, 1.4 \).

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Coupling</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalization</td>
<td>Functional, Data</td>
<td>Transitive, Non-Reflexive</td>
</tr>
<tr>
<td>Composition</td>
<td>Functional, Control, Life</td>
<td>Transitive, Non-Reflexive</td>
</tr>
<tr>
<td>Dependency</td>
<td>Functional</td>
<td>Non-Reflexive</td>
</tr>
<tr>
<td>Realization</td>
<td>Functional</td>
<td>Non-Reflexive</td>
</tr>
</tbody>
</table>

5.2 CAL model

The CAL model consists of 10 operations and 5 actions. This section formally describes and explains 4 of these operations and 1 of the actions. There are 5 action statements defined in the CAL grammar (see Appendix A), and 10 CAL operations
defined in Section 3.2. All the CAL actions were informally defined and explained in Section 3.3 using OCL-like syntax. Appendix E lists specifications of all the CAL operations with their helper libraries and one of the CAL actions, create, and its sub-actions, relabel and setweight, using TLA+ (see Section 2.6). The format used is ASCII for TLA+ as described in [50], so that it can be parsed and compiled by TLC. The operations and actions are specified as procedures using pluscal (see Section 2.6), which is based on TLA+. These specifications have been tested and verified using the TLC model checker. Since CAL operations are performed on the ADAG, we call them ADAG operations. We also calculate the compute time for the operations depend and descendants, which are used for dependency analysis.

Figure 5.1 specifies the data structure (variables) and definitions to be used in the specifications. We assume that the set of vertices (line 3) and the set of edges (line 4) already contain data, as shown in Figure 5.1. The operation isdag(), which is used in these specifications and specified in Appendix E, determines whether if the input is a DAG or not by recursively checking if there is any cycle caused by the input set, edge. This operation uses the set parents (line 1) to store the already visited parents found in a depth-first search (DFS) path in order to check if an edge contains an already visited parent (i.e. a cycle is present or not). We have defined variables as TypeInvariant (line 5) to check for type violation. There is one operator, IsEmpty (line 6), defined to check if the graph is empty or not, and it is used in specifying other operations. The set prime (line 2) contains prime numbers from 2 to 200 for labeling the graph. The labeling scheme for the DAG was described in detail in Chapter 4.
5.2.1 ADAG operations

In this section we formally specify 4 of the ADAG operations, *includes, excludes, depend* and *descendants*. We also calculate the compute time for depend and descendants.

```plaintext
define
{
  TypeInvariant ==
    \* Check if a DAG, it can only be 'DAG' (1) or 'not a DAG' (0)
    ∧ IsDAG ∈ {0, 1}
    \* weight can only be a natural number
    ∧ weight ∈ Nat
    \* depends can only have two values
    ∧ depends ∈ {0, 1}
    \* Do not exceed the length of the Set prime while labeling
    ∧ n ≤ Len(prime)
    \* LCM (Least common multiple) is a natural number
    ∧ lcm ∈ Nat
    \* Make sure EDGE_COUNT does not exceed the number of edges
    ∧ EDGE_COUNT ≤ (Len(edge) + 1)
    \* Definition of a boolean type for printing the results
    ∧ BOOL ∈ {TRUE, FALSE}

  \* Check if graph is empty. isEmpty defined as TLA+ operator
  \* that is TRUE if length of vertex is less than or equal to 0.
  \* If length of vertex is grater than 0 then it becomes FALSE.
  \*
  isEmpty == Len (vertex) ≤ 0
}
```

Figure 5.1 Global variables (data) and the define block
5.2.1.1 Includes

Figure 5.2 specifies the operation includes. It has 1 input parameter, the vertex \( iv \). It utilizes the user-defined TLA+ operator \( IsEmpty \) defined in Figure 5.1 to check if the graph is not empty. Then it checks if the set \( vertex \) contains the input vertex \( iv \).

```plaintext
procedure includes (iv)
{
    includes1: BOOL := ¬IsEmpty ∧ iv ∈ \{vertex[i][1]: i ∈ 1 .. Len(vertex)};
    print "includes(", iv, ") returns: ", BOOL;
    return;
}
```

**Figure 5.2** ADAG operation includes

5.2.1.2 Excludes

Figure 5.3 specifies the operation excludes. It has 1 input parameter, the vertex \( ev \). It also utilizes the user-defined TLA+ operator \( IsEmpty \) to check if the graph is empty. Then it checks if the set \( vertex \) does not contain the input vertex \( ev \).

```plaintext
procedure excludes (ev)
{
    excludes1: BOOL := IsEmpty ∨ ¬(ev ∈ \{vertex[i][1]: i ∈ 1 .. Len(vertex)});
    print "excludes(", ev, ") returns: ", BOOL;
    return;
}
```

**Figure 5.3** ADAG operation excludes

5.2.1.3 Depend

Figure 5.5 specifies the operation depend. It has 2 input parameters, \( v1 \) and \( v2 \), and checks for dependency between these two vertices. We assume that the graph is
already labelled as described in Section 4.1. To check for dependency we only need to
determine if the bigger label is divisible by the smaller label. The procedure
CheckDependency is used in procedure depend for computing the mod and checking for
dependency between two vertices, as shown in Figure 5.4.

```plaintext
procedure CheckDependency (v3, v4, V1, V2)
{
    CheckDependency1: if (vertex[v3][2] > vertex[v4][2] ∧
                    (vertex[v3][2] mod vertex[v4][2]) = 0)
    {
        print <<vertex[V1][1], " --> ", vertex[V2][1]>>;
        depends := 1;
    }
    else
    {
        if ((vertex[v4][2] mod vertex[v3][2]) = 0)
        {
            print <<vertex[V2][1], " --> ", vertex[V1][1]>>;
            depends := 1;
        }
    }
    return;
}
```

**Figure 5.4** ADAG sub-operation CheckDependency

The labeling scheme described in Section 4.1 labels all the leaves of the graph as
1, so we have 2 cases to check before checking for dependency. The first case (line 01 to
line 05) is when both labels of vertex v1 and v2 are not 1. In this case we simply call
CheckDependency (line 02) and there is no loop involved, so the compute time is \( O(1) \).
The other case (line 06 to line 26) is when the label of one of the vertices is 1 (a leaf). In
this case we call CheckDependency (either line 14 or 23) for each one of the immediate
parents of the vertex with label 1. If any one of these parents is dependent on the other
vertex, then the vertex (with label 1) is also dependent on the other vertex (whose label is
not 1) and the procedure returns. There is only one loop (either at line 08 or 17) involved
in this procedure, and the maximum loop number is the number of immediate parents of
procedure depend (v1, v2)
variables d = 1;
{
  Depend1: print <<"<-- Printing Dependency -->">>;
  depends := 0;
  01 if (vertex[v1][2] ≠ 1 ∧ vertex[v2][2] ≠ 1)
  { call CheckDependency (v1, v2, v1, v2);
    Depend2: if (depends = 0)
    { print <<vertex[v1][1], " and ", vertex[v2][1], " are independent">>;
    }
    05 }
  else
  { 07 if (vertex[v1][2] = 1 ∧ vertex[v2][2] ≠ 1)
    { Depend3: while (d ≤ Len(edge) ∧ depends = 0)
      { if (v1 = edge[d][2] ∧ vertex[edge[d][1]][2] ≤ vertex[v2][2])
        { v1 := edge[d][1];
          if (v1 = v2)
          { print <<vertex[v2][1], " --> ", vertex[edge[d][2]][1]>>;
            depends := 1;
          }
        }
      }
    }
    Depend4: d := d + 1;
  }
  else
  { Depend5: while (d ≤ Len(edge) ∧ depends = 0)
    { if (v2 = edge[d][2] ∧ vertex[edge[d][1]][2] ≤ vertex[v1][2])
      { v2 := edge[d][1];
        if (v2 = v1)
        { print <<vertex[v1][1], " --> ", vertex[edge[d][2]][1]>>;
          depends := 1;
        }
      }
    }
  }
  Depend6: d := d + 1;
  }
  Depend7: if (depends = 0)
  { print <<vertex[v1][1], " and ", vertex[v2][1], " are independent">>;
  }
  return;
}
the vertex with label 1, so the compute time for this case is $O(P)$, where $P$ is the number of immediate parents of the vertex with label 1. Hence, the compute time for this operation is $O(P)$ if the label of one of the vertices is 1 and $O(1)$ otherwise.

5.2.1.4 Descendants

Figure 5.6 specifies the operation \textit{descendants}. Since in this operation only one vertex (i.e. the input vertex) is compared with all other vertices, it is not optimal to call the operation \textit{depend} for all the vertices and check three cases every time for the two

```
procedure descendants (vn)
variables labelP = 1; labelD = 1; n1 = 1; n2 = 1;
{
  Descendants1: labelP := vertex[vn][2];
  print <<"<-- Printing Descendants of ", vertex[vn][1], " -->">>;
  if (labelP > 1)
    { Descendants2: while (n1 ≤ Len(vertex))
      { labelD := vertex[n1][2];
        if (labelP > labelD)
          { if (labelD = 1)
              { n2 := 1;
                Descendants3: while (n2 ≤ Len(edge))
                  { if (edge[n2][2] = vertex[n1][1]
                      ∧ labelP ≥ vertex[edge[n2][1]][2]
                      ∧ (labelP mod vertex[edge[n2][1]][2]) = 0)
                    {
                      print <<"D", vertex[n1][1]>>;
                      n2 := Len(edge);
                    };
                    Descendants4: n2 := n2 + 1;
                  };
                }; else
                  { if ((labelP mod vertex[n1][2]) = 0)
                    { print <<"D", vertex[n1][1]>>;
                    };
                  };
                Descendants5: n1 := n1 + 1;
              };
            };
          };
          Descendants6: return;
    }
}
```

\textbf{Figure 5.6} ADAG operation \textit{descendants}
vertices. Therefore, we do not use the operation \textit{depend} in this operation.

In this operation there are also two cases, but different than the two cases in the operation \textit{depend}: when the label of the vertex is 1 (line 03) i.e., a leaf, and vice versa (line 17). It iterates the set \textit{vertex} and computes the mod to check for dependency between the input vertex, \textit{vn}, and all other vertices.

The ADAG is stored in a simple list whose length is \( n \), where \( n \) is the number of vertices in the ADAG. The loop to find \textit{descendants} in Figure 5.6 executes for the length of the list, so the total number of operations is equal to the length of the list, which is the number of vertices in the DAG. Hence, the time complexity for this operation is \( O(n) \), both in the average and worst cases, where \( n \) is the number of vertices in the ADAG.

\textbf{5.2.2 CAL actions}

All CAL actions were informally defined and explained in Section 3.3 using OCL-like syntax. The most important action in CAL is the creation of the ADAG, which includes ADAG labeling and setting the weight for each vertex. The ADAG labeling scheme was described in Section 4.1, so here we formally specify and explain the operation and execution of the action \textit{create} and its sub-actions, \textit{relabel} and \textit{setweight}.

\textbf{5.2.2.1 Create}

Figure 5.7 specifies this action. Since we have already defined (with name, default weight and label, as shown in Figure 5.1) one of the input sets, \textit{vertex}, in the action \textit{create} we simply call \textit{setweight} and \textit{relabel} to set the weight and label of each vertex in the set \textit{vertex}. The purpose of this specification is not to test the performance of CAL actions, but the operation and functionality of these actions. Therefore, this action does not add any vertex or edge (to save time, they are already defined in Figure 6 as
variables), it emulates adding a vertex and an edge to the graph by using a variable EDGE_COUNT, which counts the current number of edges processed. It adds them one by one and increments the EDGE_COUNT with each addition, so the sub-actions setweight and relabel do not check beyond the number EDGE_COUNT. Hence, it implies that only EDGE_COUNT number of edges have been added.

```plaintext
procedure create ()
{
  ReLabel2: while (EDGE_COUNT ≤ Len(edge))
  {
    call setweight (EDGE_COUNT);
    ReLabel3: call relabel (edge[EDGE_COUNT][1]);
    ReLabel4: EDGE_COUNT := EDGE_COUNT + 1;
  };
  return;
}
```

**Figure 5.7** CAL action *create*

5.2.2.2 Relabel

Figure 5.8 specifies this operation. It has one input parameter vertex, *rpv*. This operation calls (line 1) the operation Label to label each vertex. The Label operation uses the labeling scheme (Section 4.1) to label a vertex and is formally specified in Appendix E. Whenever a vertex and an edge are added to the graph, all the ancestors of that vertex are relabelled using a top-first search (TFS) graph traversing technique, which is explained below. Since it traverses all the ancestors of a vertex, we also set (add) the weight of each parent depending on the relationship that it has with its children. This weight is set and defined in the setweight operation. In this way, we determine the weight of each vertex by adding the weights of all its descendants (as defined in Equation 3.1).

The loop in Figure 5.8 at line 2 iterates over the set *edge* using the count
EDGE_COUNT to emulate adding edges as described above. The procedure \textit{relabel} checks (line 3) if the input vertex \( rpv \) is the child of any vertex in the set \( edge \) (i.e. it finds all the immediate parents of the vertex \( rpv \)). If it finds any parent (line 3), then it recursively calls itself (line 4) to relabel the parent. It keeps finding till it reaches the top in a straight path. Then it backtracks to the most recent vertex that has not been explored and repeats the same steps. Since it goes to the top instead of going to the bottom, we call this method TFS.

```plaintext
procedure relabel (rpv)
variables re = 1;
{
1 RLabel5: call Label (rpv);
2 RLabel6: while (re \leq EDGE_COUNT)
{
3      if (edge[re][2] = rpv)
4        RLabel7: call relabel (edge[re][1]);
5      RLabel8: re := re + 1;
6    return;
}
}
```

\textbf{Figure 5.8} CAL sub-action \textit{relabel}

5.2.2.3 \textbf{Setweight}

Figure 5.9 specifies this operation. It has one parameter, the edge \( en \). This operation sets the weight of the edge \( en \) according to the dependency relationship defined in the configuration file in Appendix E. It sets the weight using the rule \textit{R 1.5} defined in Section 5.1.

5.3 \textbf{Formal verification using TLC}

TLC is a model checker for finding errors in TLA+ specifications. Formal specifications of the CAL model in TLA+ helped us to use TLC for verifying the CAL
procedure setweight (en)
variables edgeD = {0, 0};
{
    SetWeight1: edgeD := {edge[en][1], edge[en][2]};
    weight := 0;
    SetWeight2: if (edgeD ∈ Generalization ∨ edgeD ∈ Composition)
    {
        weight := weight + 2;
    } else
    {
        if (edgeD ∈ Dependency ∨ edgeD ∈ Realization)
        {
            weight := weight + 1;
        }
    }
    vertex[edge[en][1]][3] := vertex[edge[en][1]][3] + weight;
    return;
}

Figure 5.9 CAL sub-action setweight

model. Figure 5.10 shows a part of the main block of the CAL specifications, and Figure 5.11 shows part of the listing of TLC output. The complete CAL specifications and the TLC output appears in Appendix E. In this section, we describe the verification of the ADAG operations and CAL action and sub-actions specified in Section 5.2.2. The data used is listed in Figure 5.1.

First, TLC compiles all the files declared (included) in the TLA+ specification file and then the TLA+ file itself. In this thesis, the TLA+ specification file for CAL is CALM.tla (a full listing of this file is given in Appendix E). The TLC checks the specifications defined in the configuration file (CALM.cfg appears at the end of Appendix E). The specification to be checked is defined as Spec (temporal formula) in CALM.tla as follows:

\* In TLA+, the symbol \( \Box \) means is defined equal to and the temporal formula \( \Box F \) asserts that F is always TRUE.
\* Hence, this formula asserts that the initial state satisfies Init, and every other state satisfies Next. All the procedures called

91
The language used for the specifications in the file CALM.tla is pluscal, which is based on TLA+, so these specifications are translated to TLA+ by the TLC before compiling. Every procedure in the main block in the file CALM.tla is translated and added to the next state. TLC first checks the initial state (defined as Init in CALM.tla).

\[ \text{Spec} \equiv \text{Init} \land \Box [\text{Next}] \text{vars} \]

**Figure 5.10** Part of the main block of the CAL specifications, and the CAL data structure (ADAG) before and after annotations
TLC Version 2.0 of January 16, 2006
Model-checking
Parsing file CALM.tla
Parsing file C:\TLA\tlasany\StandardModules\Naturals.tla
Parsing file C:\TLA\tlasany\StandardModules\Sequences.tla
Parsing file C:\TLA\tlasany\StandardModules\TLC.tla
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module TLC
Finished computing initial states: 1 distinct state generated.
<< "| Start Test |" >>
<< "<-- Creating ADAG (Labeling and Setting Weight) -->" >>
<< "Vertex, label, weight --> ", 1, 2, 2 >>
<< "Vertex, label, weight --> ", 2, 3, 2 >>
<< "Vertex, label, weight --> ", 1, 6, 2 >>
<< "Vertex, label, weight --> ", 2, 3, 3 >>
<< "Vertex, label, weight --> ", 1, 6, 2 >>
<< "Vertex, label, weight --> ", 7, 5, 1 >>
<< "Vertex, label, weight --> ", 7, 5, 3 >>
<< "Vertex, label, weight --> ", 3, 7, 1 >>
<< "Vertex, label, weight --> ", 2, 21, 3 >>
<< "Vertex, label, weight --> ", 1, 42, 2 >>
<< "Vertex, label, weight --> ", 3, 35, 2 >>
<< "Vertex, label, weight --> ", 2, 105, 3 >>
<< "Vertex, label, weight --> ", 1, 210, 2 >>
<< "Vertex, label, weight --> ", 5, 9, 2 >>
<< "Vertex, label, weight --> ", 7, 45, 3 >>
<< "Vertex, label, weight --> ", 3, 315, 2 >>
<< "Vertex, label, weight --> ", 2, 3465, 3 >>
<< "Vertex, label, weight --> ", 1, 6930, 2 >>
<< "Vertex, label, weight --> ", 3, 315, 2 >>
<< "Vertex, label, weight --> ", 2, 3465, 3 >>
<< "Vertex, label, weight --> ", 1, 6930, 2 >>
<< "isempty() returns: ", FALSE >>
<< "includes(4, " returns: ", TRUE >>
<< "excludes(4, " returns: ", FALSE >>
<< "SIZE: ", 7 >>

<< "<-- Printing Dependency -->" >>
<< 1, " --> ", 3 >>
<< "<-- Printing Dependency -->" >>
<< 3, " --> ", 4 >>
<< "<-- Printing Descendants of ", 3, " -->" >>
<< "D", 4 >>
<< "D", 5 >>
<< "D", 6 >>
<< "D", 7 >>
<< "<-- Printing Descendants of ", 1, " -->" >>
<< "D", 2 >>
<< "D", 3 >>
<< "D", 4 >>
<< "D", 4 >>
<< "D", 5 >>
<< "D", 6 >>
<< "D", 7 >>
<< "<-- Printing Descendants of ", 7, " -->" >>
<< "D", 4 >>
<< "D", 5 >>
<< "D", 6 >>
<< "| End Test |" >>
Model checking completed. No error has been found.
9548 states generated, 9547 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 9547.

Figure 5.11 Part of the TLC output
and the next state (defined as Next in CALM.tla). In the initial state TLC initializes all the global variables defined in Figure 5.1. Then TLC starts generating and executing all the states defined in the next state. While executing these states, TLC checks all the type invariants (defined as \textit{TypeInvariant} in Figure 5.1) and reports an error if any one of these types is violated. TLC also checks and reports any deadlock in the states.
Chapter 6

CAL Prototypes

In this chapter we discuss two prototypes implemented in this thesis: the Constraints with Action Language (CAL) parser in detail and the tool visual CAL (VCAL) in brief. VCAL loads a UML model (as XMI) and uses the CAL parser to parse CAL files. A CAL interpreter has been implemented in Java to be used in VCAL. VCAL interprets the CAL actions and operations in a CAL file and then executes these actions and operations on the CAL data structure (ADAG). The CAL parser's UML Class Diagram, shown in Appendix D, is used in this chapter as a case study to explain and test VCAL. The test and the results obtained after running VCAL are described in Section 6.4, and the same are available as an online demonstration [63] prepared for this thesis.

6.1 CAL parser

This section gives a detailed overview of the implementation of the CAL parser. All the major classes are described, including the data structure used for storing the symbol table and the abstract syntax tree (AST). An XML file is shown for describing the output AST.

Parsing is the process of analyzing the input sequence (a string) in order to determine its structure according to a formal grammar. Parsers usually operate in two stages. The first stage is the *lexical analysis* in which the input is scanned and tokens are generated. The second stage is *syntax analysis* in which the input and tokens are
converted to a parse tree. For efficiency, these two stages are performed in parallel in the CAL parser.

6.2 Implementation details

A predictive parser [3] has been implemented manually in C++ to validate the grammar and generate the syntax tree. The rationale for using this technique is as follows: The comparison in Table 6.1 tells us that if the grammar can be transformed to fit LL(k) then LL parsing should be used, and the CAL parser implemented for this thesis shows that an LL(1) parser can be implemented manually even if the grammar is not LL(1). Although the grammar was optimized for LL parsing by removing left recursion, to keep the size of the grammar manageable left factoring was not applied to the grammar. Rather, left factoring was implemented in the CAL parser for optimization. If the grammar is more complex, then automatic parser generators should be used for LR parsing.

The CAL parser is compiled for the Microsoft Windows platform under Cygwin, but it is easily ported to other platforms (e.g. UNIX, Linux etc.). The grammar and the parser have been optimized for LL(1) parsing using techniques mentioned in Chapter 2. The state machine for lexical analysis and the syntax analysis tree are shown in Appendix B and Appendix C respectively. All the tokens for the parser are listed in Appendix A. To improve the speed of scanning, which is the most time consuming process in parsing, a CAL file is read and stored in memory as a single buffer. The parser is designed such that keeping just one token in its memory it builds the syntax tree as it scans the file buffer. Because of the length of the source code of the parser, it is not included in this thesis. A soft copy of the source code is also available online [64].
Table 6.1 Comparison of LL and LR parsing

<table>
<thead>
<tr>
<th>LL parsing</th>
<th>LR parsing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Complex</td>
</tr>
<tr>
<td>* Less number of grammars supported</td>
<td>More number of grammars supported</td>
</tr>
<tr>
<td>** Less space (table size)</td>
<td>** More space (table size)</td>
</tr>
<tr>
<td>Better parsing speed</td>
<td>Comparable parsing speed with optimization</td>
</tr>
<tr>
<td>Actions can be placed anywhere without conflict</td>
<td>Actions cannot be placed anywhere without conflict</td>
</tr>
<tr>
<td>Better error repair and reporting</td>
<td>Normal error repair and reporting</td>
</tr>
</tbody>
</table>

* Because of left factoring and left recursion
** If table driven parsers

The UML Class Diagram for the CAL parser is shown in Appendix D, and detailed documentation of the complete API (i.e. all 15 CAL parser classes) is available online [57]. Following is a description of the main features of each major class in the CAL parser.

6.2.1 Class Parser

Class Parser reads the token and analyses them according to the grammar, from left to right. For example, the first token it reads is either a comment or a package. After it reads the token package, it scans the context statements, which are of three types, attrOrAssocContext, classifierContext and operationContext. A separate procedure is implemented in this class for each non-terminal in the grammar.
6.2.2 Class OclStmt

Class OclStmt contains 1400+ lines of C++ source code and is the biggest class of the CAL parser. It implements the methods for checking the correctness of the OclStmt and also builds the AST. The OclStmt contains most of the CAL statements, as defined in the grammar given in Appendix A. The original concrete syntax of OclStmt as described in [2] contains several ambiguous production rules. These production rules have been transformed using techniques described in Chapter 2, and new rules have also been added to remove these ambiguities.

6.2.3 Class Lexer

Class Lexer performs lexical analysis of a CAL file. It reads the file in a buffer, scans it token by token and passes it to the Class Parser for syntax analysis. It processes the stored buffer and looks several characters ahead of the current input position for a pattern before a match can be found. The tokens for the CAL parser are selected such that the parsing of the file becomes easier. The main procedure, getNextToken(), is implemented as a finite state machine, which is given in Appendix C.

6.2.4 Class AST

Class AST contains procedures for building an AST. The AST starts with a root node, and children nodes are added as a linked list to the parent node, during syntax analysis of a CAL file, by the class Parser. Figure 6.1 shows the data structure used for storing the AST and is explained in the next Section.

6.2.4.1 Data structure for storing the AST

The AST is stored as a tree. Each node is labeled as a string of numbers, which increases with the number of children as shown in Figure 6.1. The label or the ID of a
node is stored as class IntList. This is the linked list for storing integers as a list. The IntList class makes it easier for a parent node to have more than 9 children. Figure 6.2 displays the output XML file generated by the class AST from a sample CAL file. The

XML tree follows the CAL grammar syntax for easy interpretation. The tags used for the XML are the tokens, which are stored in the symbol table of the CAL parser, and the attribute, name, is the value of the token, which is also called lexeme, and is scanned by the class Lexer. The CAL file used for the example in Figure 6.2 is below:

```
-- Sample file created on Sep 07, 2006 for testing CAL Parser

package simple::simpleone::simpletwo

context NamedEntities

   inv nameIsMandatory:

      self.name.isDefined and self.name <> Dude

context Person

   inv:
```
if self.sex = male then true else false endif

depackage

Figure 6.2 XML output generated by the Class AST
6.2.5 Class Symtable

Class **Symtable**, which is implemented as a *hash table*, builds the symbol table for the CAL parser and is used for storing *lexemes* and *tokens* of the language. A symbol table is built during lexical analysis and is used during syntax analysis of a CAL file. Three procedures, *lookup*, *insert* and *remove* have been implemented for this class to manipulate the symbol table. The popular *P J Weinberger C compiler* hash function (PJW) as described in [3] has been used for storing *tokens* in the symbol table. There is a chance for collision between the keys generated by any hash function. The following section describes how this collision is avoided in the implementation of the symbol table.

6.2.5.1 Collision resolution

If two keys have the same index, the corresponding value cannot be stored in the same location in the hash table. If the location is already occupied in the hash table, then we need to find another location to store this value and do it in such a way that while looking up we can find it. There are a number of collision resolution techniques, but here we describe two of the most popular ones. Table 6.2 compares these two techniques.

6.2.5.1.1 Chaining

This technique uses the concept of *buckets* in the form of a linked list to store keys in the hash table. Each slot in the hash table is a reference to the linked list. When a key collides (i.e. has the same value as the other key) with another key, then the data associated with that key is inserted in the same slot to the linked list. The data can be inserted at either end of the linked list and can be searched and removed using the list.

As shown in Figure 6.3, a linear list is used to store keys in the vertical direction, and *buckets* are used to store key values in a horizontal direction. *Edlam C* and *Tgam B*
are the two keys that give the same hash index if we use the PJW hash function, so they are stored in the linked list at the same index (hash value), which is 24. To lookup the key *Edlam C* first we go directly to index (hash value) 24 in the array and then search the linked list for the same key, which in this case is the first element of the linked list. This same technique is used in implementing the symbol table for the CAL parser.

### 6.2.5.1.2 Open addressing

This technique stores the key value directly in the hash table. Collision is resolved by searching through alternate locations in the array for the key. The key is inserted by probing for an unused slot in the hash table. To lookup a key, first the target key is compared at the index (hash value). If it is not found, then probing is used to find the next index and so on until the key is found. Figure 6.4 shows the same hash table as shown in Figure 6.3, but the keys are inserted differently. The key *Edlam C* is inserted at index (hash value) 24 and key *Tgam B* is inserted at index (hash value) 25, which is the next unused slot in the hash table.

![Figure 6.3 Collision resolution by chaining](image-url)
Figure 6.4 Collision resolution by open addressing

Table 6.2 Chaining versus open addressing

<table>
<thead>
<tr>
<th></th>
<th>Chaining</th>
<th>Open Addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple to implement</td>
<td>Difficult to implement</td>
<td></td>
</tr>
<tr>
<td>Insensitive to <em>clustering</em></td>
<td>Require better hash function to avoid <em>clustering</em></td>
<td></td>
</tr>
<tr>
<td>Table cannot fill up and doesn't show sudden increase in lookup time</td>
<td>Sudden increase in lookup time in near-full table</td>
<td></td>
</tr>
<tr>
<td>Use less space if table is sparse (more free slots)</td>
<td>Use more space if table is sparse (more free slots)</td>
<td></td>
</tr>
<tr>
<td>Use less space if the value is larger</td>
<td>Use less space if the value is small</td>
<td></td>
</tr>
<tr>
<td>Memory allocation overhead</td>
<td>No Memory allocation overhead</td>
<td></td>
</tr>
<tr>
<td>Difficult to serialize</td>
<td>Easier to serialize</td>
<td></td>
</tr>
</tbody>
</table>

Different probing techniques are used in *open addressing* hash tables. The basic difference between these techniques is the interval used for probing. Some of them are
Linear probing in which the interval between probes is fixed at 1, Quadratic probing in which the interval between probes increases linearly and is defined by a quadratic function, and Double hashing in which the interval between probes is fixed for each value. This fixed value is computed by using another hash function. The other open addressing methods are not listed here but can be found in [8].

6.2.6 Class LG

Class LG stores all the tokens defined in the CAL grammar. It also contains a method that converts and returns the token as a string for debugging and printing purposes.

6.2.7 Class List

Class List implements a linked list, which is used by different classes as their data structure. Class AST uses this class to store the list of integers. The list of integers is used to label each node in AST as described in Section 6.2.4. Class Symtable uses this class to implement buckets for collision resolution as described in Section 6.2.5. The Linked list as implemented in this thesis is shown in Figure 6.5. An iterator has also been implemented for the class List. Following is a small part of the code that deletes data stored at all the nodes from the list using iteration. First it initializes the iterator and then iterates in a loop to delete the data. The iterate() method increments the iterator and returns a pointer to the data.

```cpp
list.initIterator();
while ((nodeData = list.iterate()) != NULL)
    delete (nodeData);
```
6.3 VCAL (visual CAL)

A visual CAL parser tool, VCAL, has been implemented in Java with a graphical user interface (GUI) using the Software Widget Toolkit (SWT) [54]. An overview of VCAL is shown in Figure 6.6. We used `BigInteger` API of Java 1.5 to implement the
ADAG labeling, so JRE 1.5 is required to run VCAL. The main components of VCAL are the **CAL file**, **CAL Parser**, **ProcessXMI**, **ADAG**, **CAL Interpreter** and **GUI**. The complete VCAL implementation consists of 15 C++ classes (discussed in Section 6.2 and listed in Appendix D) for the CAL parser and 26 Java Classes for VCAL. The complete VCAL API documentation and user manual can be found in [58, 59]. In this section we briefly describe these components.

### 6.3.1 CAL parser and CAL file

VCAL uses the CAL parser to parse a CAL file and generate an AST in XML. The binaries of the CAL Parser are compiled for Microsoft Windows using cygwin. Cygwin is a collection of free software tools originally developed by Cygnus Solutions to allow various versions of Microsoft Windows to act somewhat like a Unix system. A `cygwin.dll` is included with VCAL, which is required to run the CAL parser on Microsoft Windows. The CAL parser should be able to run on other platforms (e.g. Linux, BSD and Unix), but we have not tested it on these platforms. The CAL parser is discussed in detail in Sections 6.1 and 6.2. A sample CAL file is shown in Figure 3.2 and discussed in Section 3.4 in Chapter 3.

### 6.3.2 ProcessXMI

This component loads a UML model (in XMI) into VCAL. This UML model is then processed and displayed inside the VCAL GUI. The `ProcessXMI` also queries the UML model and searches for all the classes in the Class Diagram. Then it uses pattern matching techniques, which use the XML document object model (DOM) and a Java XML parser, to map these classes with the dependency relationships (Section 5.1). After processing this information, this component passes this information as an XML file to the
ADAG component for generating the annotated (with weight and label added for each vertex) DAG using the rules R 1.1 through R 1.4, specified in Section 5.1. This DAG becomes the CAL data structure (ADAG).

6.3.3 ADAG (annotated DAG)

This component contains the CAL data structure (ADAG). It uses the labeling scheme discussed in Chapter 4 for labeling the ADAG. It also sets the weight of each vertex in the ADAG using rule R 1.5, specified in Chapter 4. It implements all the CAL actions and operations, as defined in Chapter 3, to manipulate the CAL data structure by the CAL Interpreter.

6.3.4 CAL Interpreter

This component interprets CAL actions and operations. As defined in the CAL grammar CAL has three contexts. CAL actions can only be defined in an operation context, so the CAL Interpreter only interprets an operation context in the CAL AST. The interpreter reads the AST (as XML) from the CAL parser and interprets CAL actions and operations defined in the AST. A handler class is implemented to execute (handle) these actions and operations on the CAL data structure. The interpreter also updates the ADAG, which is displayed and can be interacted inside the VCAL GUI.

6.3.5 GUI (graphical user interface)

This component is shown in Figure 6.7. The VCAL GUI is implemented using SWT. The ADAG is displayed for visualization using an open source Java API JGraph [48], which is based on the Java Abstract Window Toolkit (AWT) [56]. JGraph is a Java graph component for the display and layout of graphs. The VCAL GUI has 5 main menus (File, Compiler, View, Vdag and Help). These 5 menus have different submenus. There
are 14 toolbar items in the GUI. Some of these Toolbar items (\textit{Save DAG, Add, Delete, View Descendants, View Ancestors, View Both} and \textit{View All}) are specifically used for interacting with the ADAG. We call these toolbar items the \textit{ADAG view filters}, and they can be applied on any vertex in the ADAG. These \textit{ADAG view filters} become active only when the CAL operation \textit{view} is used in a CAL file. The use of all these \textit{ADAG view filters} is explained in the user manual of VCAL [59]. The toolbar items \textit{New, Open, Save, Print} and \textit{Run} are used as CAL file controls. For example, \textit{Run} is used to run the CAL compiler for the currently opened CAL file. All the menus and toolbar items are explained in the VCAL help file.

\textbf{Figure 6.7 VCAL GUI}
6.3.6 Running VCAL

Following is the command line to execute VCAL using Java Virtual Machine (JRE 1.5):

```
java -cp C:\VCAL\lib\swt.jar;C:\VCAL\runvcal.jar –Djava.library.path="C:\VCAL\lib"
RunVCAL
```

where

C:\VCAL\lib\swt.jar and C:\VCAL\runvcal.jar are the Java jar files. The swt.jar is the jar file for SWT and runvcal.jar is the VCAL jar file. C:\VCAL\lib is the library path. RunVCAL is the name of the root class that calls the class VCAL that contains the static main function.

6.4 Case study

In this case study, we further explain the functions of VCAL and present the test results. We chose the UML Class Diagram of the CAL parser as an example for the case study. It is not the purpose to test all the functionalities of VCAL. In this case study we want to check the usage of the CAL and test the implementation of some of the CAL actions and operations mentioned in Chapter 3. Table 6.3 lists the test matrix for this case study. The rationale for choosing this example its size, i.e., its number of classes and its detailed UML Class Diagram, as shown in Appendix G for easy comprehension.

Figure 6.8 lists a part, which is relevant to this case study, of the UML Class Diagram of the CAL parser in XMI. The complete XMI file for the CAL parser has almost 1000 lines. Appendix F lists all the classes and dependency relationships of the UML Class Diagram of the CAL parser in more detail. The XMI file for the CAL parser is part of the VCAL package [62], as one of the sample files. There are 11 Classes, 3 generalization relationships and 13 dependency relationships, as shown in Figure 6.8. Each generalization relationship has a parent and a child, and each dependency
relationship has a supplier and a client. First we load the UML model of the Class
Diagram using the XMI file for the CAL parser in VCAL, by using the menu
File→Load Model. While loading the model, VCAL also populates the CAL data.
structure (ADAG). The CAL data structure is kept in VCAL memory for efficient use. Figure 6.9 displays three CAL files with different CAL actions and operations. Now we describe how VCAL interprets and executes these three CAL files:

(a) **addClassCodeGen.cal**: We open the file *addClassCodeGen.cal* listed in Figure 6.9 (a) in the VCAL editor to run this file. There is one pre condition, three actions and three post conditions in the first operation context *addClass* in this file. The second operation context, *viewDag*, consists of only one CAL operation *view*. To run this file we press the *Run* button on the VCAL Toolbar. VCAL parses the file using the C++ CAL parser, which generates the AST in XML. The AST is then interpreted and executed by the CAL interpreter. The CAL interpreter generates the ADAG of the model. Then it creates a class *CodeGen* and links it to three classes, *AST*, *Error* and *LG*. These three classes become the parent of *CodeGen* class, or we can say that *CodeGen* becomes the descendant of these three classes in the ADAG. After this, the CAL interpreter executes the three post conditions, which check if *CodeGen* has become the descendant of these three classes. Then VCAL displays the ADAG by executing the CAL operation *view* in the context *Model::viewDag*, as shown in Figure 6.10 (a). The complete ADAG is displayed in Figure 6.7 without the *CodeGen* class. We can now apply one of the ADAG view filters, *View Ancestors*, from the toolbar on the *CodeGen*. This filter displays all the ancestors of only the *CodeGen*, as shown in Figure 6.10.

(b) **removeLinksCodeGen.cal**: As the name suggests this file, which is shown Figure in 6.9 (b), removes the links (dependencies) of one of the classes
CodeGen from the ADAG. Before removing the links we check if CodeGen has no descendants (dependents). Then we remove CodeGen from the descendants of AST, Error and LG class by removing the links from AST to CodeGen, Error to CodeGen and LG to CodeGen. In the last three post conditions in Figure 6.9 (b) we verify this operation by checking if the CodeGen is not one of the descendants of AST, Error or LG class (i.e. dependencies removed).

Table 6.3 Test matrix for the case study

<table>
<thead>
<tr>
<th>CAL action or operation</th>
<th>CAL file</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>create CodeGen()</td>
<td>addClass_CodeGen.cal</td>
<td>Class CodeGen added to the *ADAG</td>
</tr>
<tr>
<td>link AST Dependency CodeGen</td>
<td>addClass_CodeGen.cal</td>
<td>AST --&gt; CodeGen created</td>
</tr>
<tr>
<td>link Error Dependency CodeGen</td>
<td>addClass_CodeGen.cal</td>
<td>Error --&gt; CodeGen created</td>
</tr>
<tr>
<td>link LG Dependency CodeGen</td>
<td>addClass_CodeGen.cal</td>
<td>LG --&gt; CodeGen created</td>
</tr>
<tr>
<td>unlink AST CodeGen</td>
<td>removeLinks_CodeGen.cal</td>
<td>AST --&gt; CodeGen removed</td>
</tr>
<tr>
<td>unlink Error CodeGen</td>
<td>removeLinks_CodeGen.cal</td>
<td>Error --&gt; CodeGen removed</td>
</tr>
<tr>
<td>unlink LG CodeGen</td>
<td>removeLinks_CodeGen.cal</td>
<td>LG --&gt; CodeGen removed</td>
</tr>
<tr>
<td>delete CodeGen</td>
<td>deleteClass_CodeGen.cal</td>
<td>Class CodeGen deleted from the *ADAG</td>
</tr>
<tr>
<td>Dag ( Model )-&gt;view()</td>
<td>addClass_CodeGen.cal</td>
<td>The *ADAG visualized</td>
</tr>
<tr>
<td>AST-&gt;descendants()-&gt;</td>
<td>addClass_CodeGen.cal</td>
<td>True</td>
</tr>
<tr>
<td>includes(CodeGen)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error-&gt;descendants()-&gt;</td>
<td>addClass_CodeGen.cal</td>
<td>True</td>
</tr>
<tr>
<td>includes(CodeGen)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LG-&gt;descendants()-&gt;</td>
<td>removeLinks_CodeGen.cal</td>
<td>True</td>
</tr>
<tr>
<td>includes(CodeGen)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeGen-&gt;descendants()-&gt;</td>
<td>removeLinks_CodeGen.cal</td>
<td>True</td>
</tr>
<tr>
<td>size() &lt; 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodeGen-&gt;descendants()-&gt;</td>
<td>removeLinks_CodeGen.cal</td>
<td>True</td>
</tr>
<tr>
<td>size() &lt; 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not CodeGen-&gt;depend(AST)</td>
<td>removeLinks_CodeGen.cal</td>
<td>True</td>
</tr>
<tr>
<td>not CodeGen-&gt;depend(Error)</td>
<td>removeLinks_CodeGen.cal</td>
<td>True</td>
</tr>
<tr>
<td>not CodeGen-&gt;depend(LG)</td>
<td>removeLinks_CodeGen.cal</td>
<td>True</td>
</tr>
<tr>
<td>Model-&gt;excludes(CodeGen)</td>
<td>deleteClass_CodeGen.cal</td>
<td>True</td>
</tr>
<tr>
<td>Dag ( Model )-&gt;isempty()</td>
<td>deleteClass_CodeGen.cal</td>
<td>True</td>
</tr>
</tbody>
</table>

* ADAG here refers to the ADAG as visualized in the VCAL GUI shown in Figure 6.7
In this file, which is shown in Figure 6.9 (c), we delete the `CodeGen` class from the ADAG. Before deleting we check if the ADAG is not empty and the `CodeGen` has no descendants. After deleting, we confirm that the ADAG does not contain the `CodeGen` class.

```plaintext
package sample
-- Adding class
context Model::addClass ():
pre: Dag { Model }
action: create CodeGen()
action: link AST Dependency CodeGen
action: link Error Dependency CodeGen
action: link LG Dependency CodeGen
post: AST->descendants()->includes(CodeGen)
post: Error->descendants()->includes(CodeGen)
post: LG->descendants()->includes(CodeGen)
context Model::viewDag():
body: Dag { Model }->view()
endpackage

package sample
-- Removing Links
context Model::removeLinks ():
pre: Dag { Model }
pre: CodeGen->descendants()->size() < 1
action: unlink AST CodeGen
action: unlink Error CodeGen
action: unlink LG CodeGen
post: not CodeGen->depend(AST)
post: not CodeGen->depend(Error)
post: not CodeGen->depend(LG)
endpackage

package sample
-- Deleting class
context Model::deleteClass ():
pre: Dag { Model }->isempty() > 0
pre: CodeGen->descendants()->size() < 1
action: delete CodeGen
post: Model->excludes(CodeGen)
endpackage
```

**Figure 6.9** CAL files with different CAL actions and operations
Figure 6.10 VCAL GUI showing the ADAG after executing different CAL actions and operations, shown in Figure 6.9
Chapter 7

Conclusion

A software system starts changing before it is being delivered. More time is spent on changing the software system than on its development. Of all the software processes, evolution is the most expensive. There is a need to evolve the system from the beginning. To achieve this, software evolution must be integrated into the design (model) of software systems. There are two issues that our work examines and resolves. One of them is the support of software evolution in modeling languages. The other issue is the automation of model evolution. A solution is provided in the form of a modeling language CAL (Constraints with Action Language).

Finding all the dependency relationships whether direct or indirect, among UML (Unified Modeling Language) elements in a model results in a large number, making it impractical to verify, manage and analyze them manually. Due to the lack of support of software evolution management in modeling languages and tools, it is highly desirable to integrate such support into them. Our research indicates that none of the previous efforts, either commercial or academic, support automation of software evolution in a modeling language. This thesis proposed a novel approach for the automated management of software evolution in models. The proposed approach is highly portable, flexible and easy to use. OCL (Object Constraint Language) is an expression language and is guaranteed to be without side-effect. This means that the expressions do not change...
anything in the model of the software system. OCL can be used to return values. We can say that OCL expressions are like conditions that determine an action, rather than instructions that carry out an action. Using CAL, a modeler can write a constraint using the same language used for actions. Before describing our research contributions, limitations and future research possible with CAL, we first list some of the characteristics of CAL:

- Complete extension of OCL, so constraints can be used with actions. Hence, CAL combines both declarative and imperative programming paradigms;
- Provides automation for dependency analysis;
- Easy to learn. Abstracts the details so even a non-programmer in a business environment can use it effectively;
- A well-defined grammar in standard EBNF [1] form;
- Formally specified in TLA+ [50] for better comprehension, implementation and verification;
- A CAL parser implemented in C++, which can easily be integrated into other modeling tools.

The major contribution of this thesis is that we have extended OCL and defined a new language, CAL that can be used for model evolution. CAL gives a user (modeler) the ability to use constraints with actions on models, and provides the user with tools to automate model evolution. A new set of Collection directed acyclic graph (DAG) has been added to the CAL library for automating dependency analysis. CAL contains a
small set of constructs but is powerful enough to be used efficiently for typical software evolution management operations like impact analysis, correction, improvement and enhancement of models.

Since CAL has its own well-defined data structure and type system, it is independent of other modeling tool's API's, data structure and type system. This makes CAL highly interoperable, portable and flexible.

7.1 Limitations

The solution proposed and the prototype implemented in this thesis has the following limitations:

× The CAL data structure (ADAG) can only process acyclic dependency relationships among UML model elements. It is easy to remove cyclic dependency relationships, but how can we add them to the ADAG (Annotated Directed Acyclic Graph) for impact analysis? Hence, more research is required to find methods and techniques (As transformation and mapping rules) to remove and add these cyclic dependencies to the ADAG.

× VCAL loads the complete UML model, but can only process a UML model of the Class Diagram. Because in this thesis, transformation rules that are used in VCAL to transform a UML Diagram to the ADAG, are only defined for a UML Class Diagram.

× The CAL interpreter in VCAL does not provide interpretations for all the CAL instructions.

7.2 Future research

Model driven architecture (MDA) [11, 47] uses model transformation language to
transform a model from platform independent model (PIM) to platform specific model (PSM). Executable UML [9, 16] takes the concept of MDA further. It removes the PSM from MDA and directly generates code that can execute. The CAL data structure (ADAG) can be used to propagate change automatically to the transformed model or to the generated code so that when the software modeler makes a change to the model, the model compiler is able to detect which elements of the model have been modified and then can determine what minimal elements need to be recompiled or retransformed and in what order.

To use CAL for MDA or executable UML, we first need to define rules and mapping for translating PIM into the CAL data structure. As the standard for Executable UML is still Work In Progress by OMG, using CAL for executable UML can be a future research area.

CAL can also be extended for use as a full action language in MDA. To do this we need to define rules for translating CAL actions to code for different programming languages. For example, current CAL actions, operations and its data structure can be translated automatically to Java language, as is done manually in this thesis.

How to apply CAL to a UML Statechart Diagram for behavioral model evolution? A Statechart Diagram is a directed cyclic graph and can be transformed (filtered) to a DAG for dependency analysis after removing cycles from the graph. A state in a Statechart Diagram is either an operation, or a statement in some operation, in a Class Diagram, so a dependency between states of different classes in a Statechart Diagram can be mapped to the Class Diagram. We need to define these mapping rules so that we can add a dependency from a Statechart Diagram to the CAL data structure.
The implementation of a full CAL interpreter is beyond the scope of this thesis. A full CAL interpreter and compiler could be another promising research area.

This thesis has defined rules for translating a UML Class Model to the ADAG. VCAL uses these rules to load a UML model from an XMI (XML Metadata Interchange) file and translates the UML Class Model to the CAL data structure. VCAL also applies changes and filters to the ADAG. This ADAG is then saved by VCAL as an XML file. VCAL does not map this XML file of the changed ADAG to the XMI file of the UML model. This XML file needs to be translated and integrated into the UML model, so we need to define rules for this translation and integration, and we need to verify this translation and integration with the changes made in the ADAG.
CAL Grammar

TITLE: CAL (Constraints with action language) Grammar in EBNF [1]
AUTHOR: Shahid Alam
DATED: July 31, 2006
REVISION: 3.0 (January 11, 2007)

DESCRIPTION:

The grammar can be defined by a 4-tuple $G = (T, N, P, S)$ where
- $T =$ set of terminals
- $N =$ set of non-terminals
- $P =$ set of production rules
- $S =$ an element of $N$, a distinguished starting non-terminal

Section 9.3 "Concrete Syntax" and section 12.13 "Concrete Syntax of Context Declarations" in the OMG document [2] describes the grammar for OCL 2.0. It is very hard to write a parser for this grammar. As mentioned in the specification of OCL 2.0 in [2] and is quoted below:

"The grammar in this chapter might not prove to be the most efficient way to directly construct a tool . . . . Also, error correction or syntax directed editing might need hand optimized grammars".

This document describes the grammar for CAL. An optimized parser has been implemented manually to validate the grammar. This grammar is an extension to OCL 2.0 mentioned in above sections, keeping in view the abstract syntax of OCL 2.0 and action semantics [5] of UML to add actions to constraints. The grammar uses the EBNF syntax, where "|" means a choice, "?" means optional, "*" means zero or more times and "+" means one or more times. Line Comments start with "-- ". Terminator symbol is ";". Terminals are enclosed in single quotes. The start symbol is "file". It has been optimized for LL(1) parsing, to implement a predictive parser [3].

-------------------------------------------------------------------------------
--                              PRODUCTION RULES
--
file  ::= ( 'package' pathName statements 'endpackage' )+ ;
statements ::= ( statement )* ;
-------------------------------------------------------------------------------
statement ::= 'context'
| attrOrAssocContext+
| classifierContext+
| operationContext+ ;

attrOrAssocContext ::= pathName '::' id ':' type
{
    ( 'init' ')' OclStmt )+
    | ( 'drive' ')' OclStmt )+
};

classifierContext ::= pathName
{
    ( 'inv' id?':' OclStmt )+
    | ( 'def' id? ':' DefStmt )+
};

DefStmt ::= variableDeclaration '=' OclStmt
| operationName '=' OclStmt ;

operationContext ::= operationName
{
    ( 'pre' id? ':' OclStmt )+
    | ( 'post' id? ':' OclStmt )+
    | ( 'body' id? ':' OclStmt )+
    | ( 'action' id? ':' ActionStmt )+
};

operationName ::= pathName '::' id '(' parameters? ')' ':' type?
| id '(' parameters? ')' ':' type? ;

parameters ::= variableDeclaration ( ',' variableDeclaration )* ;

OclStmt ::= LogicalStmt ( ( '^^' | '^' ) MessageStmt )? ;

LogicalStmt ::= RelationalStmt
( logicalOperator RelationalStmt )* ;

RelationalStmt ::= AdditiveStmt
( relationalOperator AdditiveStmt )? ;

AdditiveStmt ::= MultiplicativeStmt
( addOperator MultiplicativeStmt )* ;

MultiplicativeStmt ::= UnaryStmt
( multiplyOperator UnaryStmt )* ;

UnaryStmt ::= ( unaryOperator PrimaryStmt
( ( '.' | '->' ) PropertyCallStmt )* )
| PrimaryStmt
( ( '.' | '->' ) PropertyCallStmt )* ;

PrimaryStmt ::= LetStmt
| IfStmt
| PropertyCallStmt
| LiteralStmt ;

PropertyCallStmt ::= id isMarkedPre ( '(' arguments? ')' )?
| id ( '[' arguments ']' )? isMarkedPre?
| id ( '(' arguments ')' )
| id
'(' ( variableDeclaration
     ( ',' variableDeclaration )? '|
     )? OclStmt
  ')' |
pathName ( '(' arguments? ')' )?
| 'iterate'
  '(' ( variableDeclaration ';' )?
       variableDeclaration '|
       OclStmt
  ')' ;
arguments ::= OclStmt ( ',' OclStmt )* ;
LiteralStmt ::= CollectionLiteralStmt
                 | TupleLiteralStmt
                 | PrimitiveLiteralStmt ;
CollectionLiteralStmt ::= CollectionTypeIdentifier
                         '{' CollectionLiteralParts? '}' ;
CollectionTypeIdentifier ::= 'Set'
                           | 'Dag'
                           | 'Bag'
                           | 'Sequence'
                           | 'Collection'
                           | 'OrderedSet' ;
CollectionLiteralParts ::= CollectionLiteralPart
                         ( ',' CollectionLiteralPart )* ;
CollectionLiteralPart ::= CollectionRange | OclStmt ;
CollectionRange ::= OclStmt '..' OclStmt ;
TupleLiteralStmt ::= 'Tuple' '{' variableDeclarations '}' ;
variableDeclaration ::= id (':' type)? ( '=' OclStmt )? ;
tupleType ::= 'TupleType' '{' variableDeclarations '}' ;
variableDeclarations ::= variableDeclaration (',' variableDeclaration )* ;
PrimitiveLiteralStmt ::= NumericLiteralStmt
                          | StringLiteralStmt
                          | BooleanLiteralStmt ;
NumericLiteralStmt ::= number ;  -- integer | real
StringLiteralStmt ::= '"' string '"' ;
BooleanLiteralStmt ::= 'true' | 'false' ;
LetStmt ::= 'let' variableDeclaration LetStmtSub ;
LetStmtSub ::= ',' variableDeclaration LetStmtSub
             | 'in' OclStmt ;
MessageStmt ::= id ( '(' MessageArguments? ')' ) ;
MessageArguments ::= MessageArg (',' MessageArg)* ;
MessageArg ::= OclStmt
              | ( '?' (':' type)? ) ;
IfExp ::= 'if' OclStmt
'then' OclStmt
'else' OclStmt
'endif';

---------------------------------------------------------------
-- RULES ADDED FOR CAL
--
-- create, delete and addattr library function will increment the global
-- variable CHANGE_RATE whenever any attribute or operation is added or deleted
-- from the class / object. It is the number of classes changed and can be used
-- to calculate the %age of classes that changed from one version to the next,
-- which is called change rate. Similarly create and delete will
-- increment / decrement the global variables CLASSES_ADDED / CLASSES_REMOVED
-- respectively to calculate the growth rate which is the difference between
-- added and removed classes / objects [4]
--
---------------------------------------------------------------

ActionStmt ::= createStmt
              | deleteStmt
              | addAttrStmt
              | createLinkStmt
              | deleteLinkStmt;

createStmt ::= 'create' operation
              | 'create' operation 'to' pathName;

deleteStmt ::= 'delete' id
              | 'delete' id 'of' pathName;

addAttrStmt ::= 'addattr' id
               | id 'to' pathName;

createLinkStmt ::= 'link' pathName association pathName;

deleteLinkStmt ::= 'unlink' pathName pathName;

operation ::= visibility* id '(' parameters* ')';

---------------------------------------------------------------
-- To create a DAG, so that the dependent objects should also
-- get deleted, and the global variables mentioned above for
-- change and growth rate should also be updated
-- deleteStmt deletes both operation and attr (id can be operation / attr)
--
---------------------------------------------------------------

keyword ::= and | context | def | derive | else
           | endif | endpackage | false | if | implies
           | in | init | inv | let | null | action
           | not | or | package | post | pre | body
           | then | true | xor | TUple | TupleType
           | public | private | protected
           | create | delete | addAttr
           | link | unlink
           | of | to;
LOP := 'and' | 'or' | 'xor' | 'implies' ;
ROP := '=' | '<' | '>' | '<=' | '>=' | '<>' ;
NOT := 'not' ;
MUL := '*' ;
DIV := '/' ;
PLUS := '+' ;
MINUS := '-' ;
DAOP := '-->' | '.' ;
MSGOP := '^' | '^^' ;
RANGEOP := ',' ;
CTID := 'Set'
  | 'Dag'
  | 'Bag'
  | 'Sequence'
  | 'Collection'
  | 'OrderedSet' ;
STRING := '"' character* '"' ;
LBRKT1 := '(' ;
RBRKT1 := ')' ;
LBRKT2 := '[' ;
RBRKT2 := ']' ;
LBRKT3 := '{' ;
RBRKT3 := '}' ;
PIPE := '|' ;
ATPTRE := '@pre' ;
TUPLE := 'Tuple' ;
TUPLET := 'TupleType' ;
PACKAGE := 'package' ;
ENDPACKAGE := 'endpackage' ;
CONTEXT := 'context' ;
INIT := 'init' ;
DRIVE := 'drive' ;
INV := 'inv' ;
DEF := 'def' ;
LET := 'let' ;
PRE := 'pre' ;
POST := 'post' ;
BODY := 'body' ;
ACTION := 'action' ;
IF := 'if' ;
THEN := 'then' ;
ELSE := 'else' ;
ENDIF := 'endif' ;
TO := 'to' ;
OF := 'of' ;
VIS := 'public'
  | 'private'
  | 'protected' ;
CAL Parser's Finite State Machine for Lexical Analysis

Substate Machine for STATE '0'
Appendix C

CAL Parser's Syntax and Parsing Tree

Main Syntax tree for CAL Parser
Syntax tree for DefStmt in CAL Parser
Syntax tree for PrimaryStmt in CAL Parser
Appendix E

CAL Specifications (ASCII Version Using pluscal, Translation not Shown) in TLA+ with Configuration File and TLC Output

\*--------------------------------------------------------
\*
\* Filename: CALM.tla
\* Dated: 10, February, 2007
\* Author: Shahid Alam (salam@sce.carleton.ca)
\*
\* Description: Specification of CAL operations in ASCII format using TLA+ for verification. Specification is in the form of algorithms for each operation. These specifications have been translated to TLA+, compiled, run and verified by the TLC (TLA+ Model Checker).
\*
\* The following CAL operations and sub-operations are specified:
\* isempty (), isdag (), includes (vertex), excludes (vertex), depend (vertex1, vertex2),
\* descendants (vertex), flatten (), view (),
\* CheckCycle (vertex1, vertex2),
\* CheckDependency (vertex1, vertex2),
\* LCM (number1, number2), Label (vertex)
\*
\* The following CAL actions and sub-actions are specified:
\* create (), relabel (vertex), setweight (edge)
\*--------------------------------------------------------

---------------------- MODULE CALM ----------------------

\* Built in libraries for functions for Natural numbers, Sets and TLC operators (print etc) defined for TLA+
\* EXTENDS Naturals, Sequences, TLC
\*
\* Sets of all dependency relationships (as edges)
\*
\* CONSTANT Generalization, Composition, Dependency, Realization
\*
\*--algorithm CALM
\{
\variables
  \* Set of already visited parents to check for cycle in the graph
  parents = << >>;
  \* Set of prime numbers from 2 – 200 for labeling
  prime = << 2,3,5,7,9,11,13,17,19,23,29,31,37,41,47,53,59,61,71,73,79,83,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,199 >>;
  \* Set of vertices in the graph with name, default label and default weight
  vertex = << <<1,1,0>>, <<2,1,0>>, <<3,1,0>>, <<4,1,0>>, <<5,1,0>>, <<6,1,0>>, <<7,1,0>> >>;
  \* Set of edges in the graph
\}
\[ edge = \langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 7, 5 \rangle, \langle 4, 7 \rangle, \langle 3, 7 \rangle, \langle 5, 6 \rangle \];

IsDAG = 1;
weight = 0;
depends = 0;
n = 1;
lcm = 0;
EDGE_COUNT = 1;
BOOL = FALSE;

\textbf{define}

\{ 
\textbf{TypeInvariant} ==
  /* Check if a DAG, it can only be 'DAG' (1) or 'not a DAG' (0)
  */ IsDAG \in \{0, 1\}
  /* weight can only be a natural number */ weight \in \mathbb{Nat}
  /* depends can only have two values */
  depends \in \{0, 1\}
  /* Do not exceed the length of the Set prime while labeling
  */ n \leq \text{Len(prime)}
  /* LCM (Least common multiple) is a natural number */
  lcm \in \mathbb{Nat}
  /* Make sure EDGE_COUNT does not exceed the number of edges */
  EDGE_COUNT \leq (\text{Len(edge)} + 1)
  /* Definition of a boolean type for printing the results */
  BOOL \in \{\text{TRUE}, \text{FALSE}\}
\}

/* Check if graph is empty. IsEmpty defined as TLA+ operator
  * that is TRUE if length of vertex is less than or equal to 0.
  * If length of vertex is greater than 0 then it becomes FALSE.
  */
IsEmpty == \text{Len (vertex)} \leq 0

/* Using the TLA+ operator defined in define, check if graph
  * is empty or not.
  */
\textbf{procedure isempty ()}

{ 
IsEmpty1: BOOL := IsEmpty;
print <<"isempty() returns: ", BOOL>>, return;
}

/* If graph is not empty and iv is in the Set vertex then
  * BOOL is TRUE else BOOL is FALSE
  */
\textbf{procedure includes (iv)}

{ 
Includes1: BOOL := ~IsEmpty \land iv \in \{vertex[i][1]: i \in 1 .. \text{Len(vertex)}\};
print <<"includes(" iv, ") returns: ", BOOL>>, return;
}

/* If graph is empty or ev is not in the Set vertex then
  * BOOL is TRUE else BOOL is FALSE
  */
\textbf{procedure excludes (ev)}

{ 
Excludes1: BOOL := IsEmpty \lor ~\{ev \in \{vertex[i][1]: i \in 1 .. \text{Len(vertex)}\};
print <<"excludes(" ev, ") returns: ", BOOL>>, return;
}

/* Computes size of the Set vertex using the library
  * function \text{Len} from Sequences. */
procedure size ()
{
    Size1: print <<"SIZE: ", Len(vertex)>>;
    return;
}

procedure isdag ()
{
    Isdag1: IsDAG := 1;
    /* Add the first parent to already visited parents Set and check for cycle. */
    parents := <<edge[1][1]>> \o parents;
    call CheckCycle (edge[1][1], edge[1][2]);
    return;
}

procedure CheckCycle (parent, child)
variables cycle = 1;
{
    /* If child which is new parent is in the Set of already visited parents then a cycle is detected. */
    /* */
    CheckCycle1: if (child \in {parents[i]: i \in 1 .. Len(parents)})
    {
        print <<"Cycle @ Vertex: ", child>>;
        IsDAG := 0;
    }
    else
    {
        /* Iterate over the Set edge for the path */
        /* */
        CheckCycle2: while (IsDAG = 1 \ cycle \leq Len(edge))
        {
            /* If child is the new parent */
            if (child = edge[cycle][1])
            {
                BOOL := TRUE;
                /* If child which is new parent is not in the Set parents */
                /* then add it to the already visited parents Set */
                /* and check for cycle. */
                if (~(child \in {parents[i]: i \in 1 .. Len(parents)}))
                {
                    parents := <<child>> \o parents;
                }
            }  
        }  
    } 
} 
*/
Next edge in the path

call CheckCycle (edge[cycle][1], edge[cycle][2]);

else

BOOL := FALSE;

CheckCycle4: cycle := cycle + 1;

if (BOOL = FALSE)

\* Remove the last parent added from Set parents

parents := Tail (parents);

CheckCycle5: return;

\* Check for dependency between two vertices.

\* There can be two cases:
\* 1. Both labels are not 1.
\* \* Compute time O(1).
\* 2. One of the label is 1.
\* \* Compute time O(P), where P is the number of
\* \* immediate parents of the vertex with label 1.

procedure depend (v1, v2)
variables d = 1;

Depend1: print "<-- Printing Dependency -->";

depends := 0;

\* None of the label is 1

if (vertex[v1][2] # 1 /\ vertex[v2][2] # 1)

call CheckDependency (v1, v2, v1, v2);

Depend2: if (depends = 0)

print "vertex[v1][1], " and ", vertex[v2][1], " are independent";

Depend3: while (d \leq Len(edge) /\ depends = 0)

\* edge first element is parent of v1 (i.e v1 is the second element)

if (v1 = edge[d][2] /\ vertex[edge[d][1]][2] \leq vertex[v2][2])

v1 := edge[d][1];

\* Immediate parent

if (v1 = v2)

print "vertex[v2][1], " --> " , vertex[edge[d][2]][1]";

depends := 1;

138
else
   { call CheckDependency (v1, v2, edge[d][2], v2);
    }
};
Depend4: d := d + 1;
);
else
   { /* v2's label is 1.
     * Iterate all the parents of v2 and check for dependency
     */
Depend5: while (d \leq Len(edge) \&\& depends = 0)
   {
     /* edge first element is parent of v2 (i.e v2 is the second element)
      if (v2 = edge[d][2] \&\& vertex[edge[d][1]][2] \leq vertex[v1][2])
      { /* Immediate parent
         v2 := edge[d][1];
         if (v2 = v1)
         { print <<vertex[v1][1], " --> ", vertex[edge[d][2]][1]>>;
          depends := 1;
        }
        else
         { call CheckDependency (v1, v2, v1, edge[d][2]);
          }
      ;
      Depend6: d := d + 1;
      }
    ;
Depend7: if (depends = 0)
   { print <<vertex[v1][1], " and ", vertex[v2][1], " are independent">>;
    }
   return;
   }
/* Compute (v3 Mod v4) to check for dependency between two vertices.
 * If label is 1 then immediate parents are used for checking dependency.
 * So V1 and V2 are the original vertices used by print function.
 */
procedure CheckDependency (v3, v4, V1, V2)
{ /* If v3's label is greater than v4's label
   * then v3 mod v4
   */
CheckDependency1: if (vertex[v3][2] > vertex[v4][2] \&\&
  (vertex[v3][2] \% vertex[v4][2]) = 0)
   { print <<vertex[V1][1], " --> ", vertex[V2][1]>>;
    depends := 1;
    }
else
   { /* If v4's label is greater than v3's label
      * then v4 mod v3
      */
if ((vertex[v4][2] \% vertex[v3][2]) = 0)
   { print <<vertex[V2][1], " --> ", vertex[V1][1]>>;
    depends := 1;
    };
};
Compute LCM (least common multiple) of two numbers using the following formula:
\[
gcd = \text{GCD} (N1, N2) \text{ using Euclid's algorithm}
\]
\[
lcm = gcd \times ((N1 / gcd) \times (N2 / gcd))
\]

procedure LCM (N1, N2)
variables n3 = 0; n4 = 0;
{
    LCM1: lcm := 0;
    n3 := N1;
    n4 := N2;
    LCMGCD1: while (n3 # 0)
    {
        if (n3 < n4)
        {
            \* swap n3 and n4
            n3 := n4 || n4 := n3;
        }
        LCMGCD2: n3 := n3 - n4;
    }
    if (n4 # 0)
    {
        lcm := n4;
        LCM2: lcm := lcm * ((N1 \div lcm) \times (N2 \div lcm));
    }
    LCM3: return;
}

procedure Label (pv)
variables la = 1; largest = 1; label = 1;
{
    Label1: label := vertex[pv][2];
    \* Iterate over the Set edge till EDGE_COUNT.
    \* See procedure create for definition of EDGE_COUNT.
    \* Label2: while (la \leq EDGE_COUNT)
    {
        \* Get all the children of the vertex
        \* Label3: if (edge[la][1] = pv)
        {
            \* Compute LCM of all the children one by one.
            \* call LCM (label, vertex[edge[la][2]][2]);
            Label4: label := lcm;
            \* Store the largest label of the children.
            \* Label5: if (vertex[edge[la][2]][2] > largest)
            {
                largest := vertex[edge[la][2]][2];
            }
        }
    }
    Label6: la := la + 1;
}
\* Compare the largest label with the LCM (label) of the vertex. If it's equal then multiply with the next prime.
\*
Label7: if (label = largest)
{ 
  label := label * prime[n];
  if (n \geq Len(prime))
  { 
    n := 0;
  }
  Label8: n := n + 1;
};
\*
\* Assign and store the computed label int the Set vertex
\*
Label9: vertex[pv][2] := label;
print <<"Vertex, label, weight --> ", vertex[pv][1],
  vertex[pv][2], vertex[pv][3]>>;
return;
}
\*
\* Recursive Label Procedure.
\* It relabels all the ancestors of the vertex.
\*
procedure relabel (rpv)
variables re = 1;
{
\*
  \* Label the vertex
  \*
  RLabel5: call Label (rpv);
  \*
  \* Iterate over the Set edge till EDGE_COUNT.
  \* See procedure create for definition of EDGE_COUNT.
  \*
  RLabel6: while (re \leq EDGE_COUNT)
  { 
    \* Label all the ancestors recursively
    \*
    if (edge[re][2] = rpv)
    { 
      RLabel7: call relabel (edge[re][1]);
    }
    RLabel8: re := re + 1;
  }
  return;
}
\*
\* Creating ADAG. Add edges to the ADAG by labeling and setting weight of each parent vertex in the Set edge.
\* It calls recursive function relabel to label and set weight of all the vertices.
\*
procedure create ()
{
\*
  \* Initialize of EDGE_COUNT which gives the current count of edges added to the ADAG.
  \*
  ReLabel2: while (EDGE_COUNT \leq Len(edge))
  { 
    \* Set the weight for each edge
    \*
    call setweight (EDGE_COUNT);
    \*
\* Label and set weight for each vertex in the edge.
\* Other vertices are already assigned default label (1) and default weight (0).
\* Relabel3: call relabel (edge[EDGE_COUNT][1]);
\* Relabel4: EDGE_COUNT := EDGE_COUNT + 1;
};
return;
}

\* Sets the weight according to the dependency relationship,
\* by checking in which Sets of dependencies (defined in configuration file) the edge belongs to.
\*
procedure setweight (en)
variables edgeD = (0, 0);
{
    SetWeight1: edgeD := (edge[en][1], edge[en][2]);
    weight := 0;
    \* Check if edge is contained in the Set Generalization or Composition.
    \* SetWeight2: if (edgeD \in Generalization \or edgeD \in Composition)
    { weight := weight + 2; }
    else
    {
        \* Check if edge is contained in the Set Dependency or Realization.
        \* SetWeight3: if (edgeD \in Dependency \or edgeD \in Realization)
        { weight := weight + 1; }
    }
    \* Assign the computed weight to the vertex
    \* vertex[edge[en][1]][3] := vertex[edge[en][1]][3] + weight;
    return;
}

\* Print As Set.
\* It iterates the Set vertex and prints each vertex name, label and weight.
\*
procedure flatten ()
variables f = 1;
{
    Flatten2: print <<"<-- Printing Graph Vertices with Label and Weight --> ">>;
    \* Iterate over the Set vertex and print vertex name, label and weight.
    \* Flatten3: while (f \leq Len(vertex))
    { print <<vertex[f][1], vertex[f][2], vertex[f][3]>>;
        f := f + 1;
    }
    return;
}

\* Computes and prints all the descendants of the vertex.
\* It iterates through the Set vertex and compute the \* mod of the input vertex vn with all other vertices.
/* if the mod is 0 then the vertex is descendant of vertex vn. */
/* The compute time is O(N), where N is the number of vertices */
/* in the Set vertex. */

procedure descendants (vn)
variables labelP = 1; labelD = 1; n1 = 1; n2 = 1;
{
    Descendants1: labelP := vertex[vn][2];
    print <<"-- Printing Descendants of ", vertex[vn][1], " -->">;
    /* Check if any descendants (i.e the vertex is not a leave)
    */
    if (labelP > 1)
    {
        /* Iterate over the Set vertex */
        Descendants2: while (n1 \leq Len(vertex))
        {
            labelD := vertex[n1][2];
            if (labelP > labelD)
            {
                /* If label to be checked is 1 then check all
                */
                /* it's parents label. */
                if (labelD = 1)
                {
                    n2 := 1;
                    /* Iterate over the Set edge */
                    Descendants3: while (n2 \leq Len(edge))
                    {
                        /* Get all the parents of the vertex only if
                        */
                        /* labelP (vertex's label) is greater than the
                        */
                        /* parent's label and (labelP) mod (parent's label)
                        */
                        /* is 0 then vertex with labelD is a descendant.
                        */
                        if (edge[n2][2] = vertex[n1][1]
                            /* labelP \geq vertex[edge[n2][1]][2]
                            */
                            /* (labelP % vertex[edge[n2][1]][2]) = 0 */
                            
                            
                            print <<"D", vertex[n1][1]>>;
                            n2 := Len(edge);
                        );
                        Descendants4: n2 := n2 + 1;
                    );
                    /* If (labelP) mod (next vertex's label) is 0
                    */
                    /* then next vertex is a descendant. */
                    else
                    {
                        if ((labelP % vertex[n1][2]) = 0)
                        {
                            print <<"D", vertex[n1][1]>>;
                        );
                    );
                    Descendants5: n1 := n1 + 1;
                );
            );
        Descendants6: return;
    }
    /* It prints the complete (vertices and edges) graph */
procedure view ()

variables p = 1;
{
    PrintAll1: print <<"-- Printing Complete Graph -->">>;
    print <<"-- Vertices -->">>;
    \*
    \* Iterate over the Set vertex and prints the vertex name
    \*
    PrintAll2: while (p \leq Len(vertex))
    { 
        print <<vertex[p][1]>>;
        p := p + 1;
    };
    print <<"-- Edges -->">>;
    p := 1;
    \*
    \* Iterate over the Set edge and prints the source and the target for each edge.
    \*
    PrintAll3: while (p \leq Len(edge))
    { 
        print <<edge[p][1], edge[p][2]>>;
        p := p + 1;
    };
    return;
}

\* Main Algorithm, Check All Procedures
\*
{
    MainPrintStart:
    print <<"|---------------------|">>;
    print <<"|      Start Test   |
    print <<"|---------------------|">>;
    Main1: call isdag ();
    MainIf: if (IsDAG = 0)
    { 
        print <<"|----------|">>;
        print <<"| Not a DAG |">>;
        print <<"|----------|">>;
    };
    else 
    { 
        MainPrint: print <<"<-- Creating ADAG (Labeling and Setting Weight) -->">>;
        Main2: call create ();
        Main3: call view ();
        Main4: call isempty ();
        Main5: call includes (4);
        Main6: call excludes (4);
        Main7: call size ();
        Main8: call flatten ();
        Main9: call depend (3, 1);
        Main10: call depend (3, 4);
        Main11: call descendants (3);
        Main12: call descendants (1);
        Main13: call descendants (7);
    };
    MainPrintEnd:
    print <<"|---------------------|">>;
    print <<"|       End Test     |
    print <<"|---------------------|">>;
}
}
Configuration File

\*--------------------------------------------------------
\* Filename: CALM.cfg
\* Dated: 10, February, 2007
\* Author: Shahid Alam
\* Description: Configuration file for CALM.tla. It
\* specifies all the dependency sets
\* that are used in CALM.tla for
\* assigning self weights to vertices.
\* It also specifies the types and
\* specification to be checked and tested.
\*-------------------------------------------------------

CONSTANT
Generalization = { {1, 2}, {2, 3}, {7, 4} } \* weight = 2
Composition = { {5, 6} } \* weight = 2
Dependency = { {2, 4}, {3, 5}, (7, 5) } \* weight = 1
Realization = { {3, 7} } \* weight = 1

INVARIANT TypeInvariant
SPECIFICATION Spec

TLC Run for Model Checking

C:\thesis\CALModel>java tlc.TLC -cleanup -difftrace CALM.tla
TLC Version 2.0 of January 16, 2006
Model-checking
Parsing file CALM.tla
Parsing file C:\TLA\tlasany\StandardModules\Naturals.tla
Parsing file C:\TLA\tlasany\StandardModules\Sequences.tla
Parsing file C:\TLA\tlasany\StandardModules\TLC.tla
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module TLC
Semantic processing of module CALM
Finished computing initial states: 1 distinct state generated.
<< "|---------------------|
<< |      Start Test     |
<< |---------------------|
<< "-- Creating ADAG (Labeling and Setting Weight) -->" >>
<< "Vertex, label, weight --> ", 1, 2, 2 >>
<< "Vertex, label, weight --> ", 2, 3, 2 >>
<< "Vertex, label, weight --> ", 1, 6, 2 >>
<< "Vertex, label, weight --> ", 2, 3, 3 >>
<< "Vertex, label, weight --> ", 1, 6, 2 >>
<< "Vertex, label, weight --> ", 7, 5, 1 >>
<< "Vertex, label, weight --> ", 7, 5, 3 >>
<< "Vertex, label, weight --> ", 3, 7, 1 >>
<< "Vertex, label, weight --> ", 2, 21, 3 >>
<< "Vertex, label, weight --> ", 1, 42, 2 >>
<< "Vertex, label, weight --> ", 3, 35, 2 >>
<< "Vertex, label, weight --> ", 2, 105, 3 >>
<< "Vertex, label, weight --> ", 1, 210, 2 >>
<< "Vertex, label, weight --> ", 5, 9, 2 >>
<< "Vertex, label, weight --> ", 7, 45, 3 >>
<< "Vertex, label, weight --> ", 3, 315, 2 >>
<< "Vertex, label, weight --> ", 2, 3465, 3 >>
<< "Vertex, label, weight --> ", 1, 6930, 2 >>
<< "Vertex, label, weight --> ", 3, 315, 2 >>
<< "Vertex, label, weight --> ", 2, 3465, 3 >>
<< "Vertex, label, weight --> ", 1, 6930, 2 >>
<< "-- Printing Complete Graph -->" >>
"-- Vertices -->"
1
2
3
4
5
6
7

"-- Edges -->"
1, 2
2, 3
2, 4
7, 5
7, 4
3, 5
3, 7
5, 6

"isempty() returns: ", FALSE
"includes(" , 4, " ) returns: " , TRUE
"excludes(" , 4, " ) returns: " , FALSE
"SIZE: " , 7

"--- Printing Graph Vertices with Label and Weight --> "
1, 6930, 2
2, 3465, 3
3, 315, 2
4, 1, 0
5, 9, 2
6, 1, 0
7, 45, 3

"--- Printing Dependency -->"
1, " --> ", 3

"--- Printing Dependency -->"
3, " --> ", 4

"--- Printing Descendants of ", 3, " -->"
"D", 4
"D", 5
"D", 6
"D", 7

"--- Printing Descendants of ", 1, " -->"
"D", 2
"D", 3
"D", 4
"D", 5
"D", 6
"D", 7

"--- Printing Descendants of ", 7, " -->"
"D", 4
"D", 5
"D", 6

"|---------------------|
<table>
<thead>
<tr>
<th>End Test</th>
</tr>
</thead>
</table>

Model checking completed. No error has been found.
Estimates of the probability that TLC did not check all reachable states because two distinct states had the same fingerprint:
calculated (optimistic): 5.175439070359555E-16
based on the actual fingerprints: 1.8406996829771002E-12
9548 states generated, 9547 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 9547.
Appendix F

UML Class Diagram in XMI for the Case Study

This appendix lists the UML Class Diagram in XMI for the case study described in Chapter 6. An open source UML CASE tool StarUML [60] is used to generate the XMI from the UML Class Diagram of the CAL parser. We have not listed the complete XMI. Only the classes and their relationships are shown.

```xml
<XMI xmi.version="1.1" xmlns:UML="bref://org.omg/UML/1.3"
<?xml version="1.0" encoding="UTF-8"?>
timestamp = "Wed Feb 21 10:27:58 2007">
<XMI.content>
<UML:Model xmi.id="UMLProject.1">
<UML:Namespace.ownedElement>
<UML:Model xmi.id="UMLModel.2" name="CALParser" visibility="public"
isSpecification="false" namespace="UMLProject.1" isRoot="false"
isLeaf="false" isAbstract="false">
<UML:Namespace.ownedElement>
<UML:Class xmi.id="UMLClass.52" name="ListNode" visibility="public"
isSpecification="false" namespace="UMLModel.2"
clientDependency="UMLDependency.403" supplierDependency="UMLDependency.435"
isRoot="false" isLeaf="false" isAbstract="false" generalization="UMLGeneralization.400"
specialization="UMLGeneralization.399 UMLGeneralization.454" participant="UMLAssociationEnd.137 UMLAssociationEnd.140"
isActive="false">
</UML:Class>
<UML:Class xmi.id="UMLClass.68" name="List" visibility="public"
isSpecification="false" namespace="UMLModel.2"
clientDependency="UMLDependency.413 UMLDependency.435"
supplierDependency="UMLDependency.403 UMLDependency.439"
isRoot="false" isLeaf="false" isAbstract="false" generalization="UMLGeneralization.399 UMLGeneralization.401"
UMLGeneralization.454" specialization="UMLGeneralization.400 UMLGeneralization.438" participant="UMLAssociationEnd.136"
isActive="false">
</UML:Class>
<UML:Class xmi.id="UMLClass.104" name="TreeNode" visibility="public"
isSpecification="false" namespace="UMLModel.2"
clientDependency="UMLDependency.402 UMLDependency.436 UMLDependency.437"
supplierDependency="UMLDependency.402 UMLDependency.404"
isRoot="false" isLeaf="false" isAbstract="false" generalization="UMLGeneralization.401 UMLGeneralization.438" specialization="UMLGeneralization.401 UMLGeneralization.455" participant="UMLAssociationEnd.139"
isActive="false">
</UML:Class>
<UML:Class xmi.id="UMLClass.141" name="AST" visibility="public"
isSpecification="false" namespace="UMLModel.2"
clientDependency="UMLDependency.402 UMLDependency.437"
supplierDependency="UMLDependency.402 UMLDependency.437"
isRoot="false" isLeaf="false" isAbstract="false" generalization="UMLGeneralization.401 UMLGeneralization.455" isActive="false">
</UML:Class>
<UML:Class xmi.id="UMLClass.171" name="Symtable" visibility="public"
isSpecification="false" namespace="UMLModel.2"
clientDependency="UMLDependency.405 UMLDependency.439"
supplierDependency="UMLDependency.413 UMLDependency.442"
</UML:Class>
</UML:Model>
</XMI.content>
</XMI>
```
References


