Model Translation:
A UML-based specification technique and active implementation approach

A thesis submitted to
The University of Kent at Canterbury
for the degree of
Doctor of Philosophy

David H Akehurst
December 2000

To my Dad

Abstract

Many software applications involve models of data that are manipulated by the application. There is often a need to transform (or translate) the data from one model, into another in which the data is differently structured. In addition, there is an increasing requirement to pass data between different applications, which invariably have different formats for their data models.

Both of these issues require a translation of the modelled data from one form to another. The process of translating a model from one form to another is known as *model transformation* or *model translation*.

The literature on model transformation includes a number of techniques for specifying transformations. However, the majority of these techniques are grammar-based specifications, many of which use a textual grammar, although some make use of graphical (graph) grammars. These subsequently lead to a monolithic one-step implementation process that performs the transformation.

This thesis addresses two issues that are related to the area of model transformation. Firstly, it addresses the need for a standard notation that can be used for writing model translator specifications. Secondly, a technique for implementing model translators is developed that actively performs the transformation. Rather than a single step process, that must be executed every time the source model changes, the active implementation approach presented performs a continuous translation updating the target model every time a change is made to the source model.

The specification technique makes use of the standardised Unified Modelling Language (UML) and Object Constraint Language (OCL) for specifying a transformation relationship between two object-oriented models, each of which is also specified using UML and OCL.

The implementation approach uses an event-based version of the observer pattern enabling the construction of translator to be formed from a number of *mini-translator* parts, each of which monitors a small set of components. These mini-translators act upon events generated by the model components and update the transformed components to reflect the changes.

The specification and implementation techniques described can be applied to many problem areas. In particular this thesis discusses their application to Multiple View Visual Languages (i.e. the UML itself) and automatic performance model generation.

Acknowledgements

Many thanks to my supervisors Dr. Gill Waters and Prof. Peter Linington for their support and invaluable advice throughout the duration of the research and writing of this thesis. Especially, my thanks to Peter for showing me how to form a coherent argument from my initial collection of ideas.

Thanks also to the many others who have proof read various extracts and discussed or argued about the ideas presented. In particular, thanks to Dr. John Derrick who read my first attempt at putting the whole thing together. Thanks to him, my wife, and supervisors for their advice on suitable writing styles and techniques for presenting ideas in a written format.

Acknowledgements also to British Telecom for funding the Permabase project and my initial introduction to the world of academic research. Thanks to all the members of that project for the discussions held during it; in particular, to Andrew Symes with whom I worked closely throughout the duration of the project (and is still speaking to me).

Thanks to both my parents for their support, encouragement and tutelage during my early years, without which I would never have got to the point of being able to attempt a doctoral degree.

Thanks to Sally Fincher for her help in getting around the “features” of the word-processor I used to type up this thesis; and to Behzad Bordbar for checking my maths.

Finally, thanks to my family – Kerry, Luc and Paul – for their tolerance and patience during the final period of writing up.

Declaration

The content of this thesis is a product of the author's original work except where explicitly stated as otherwise.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modelling is a principal exercise in software engineering and development and one of the current practices is object-oriented modelling. The Object Management Group (OMG) has defined a standard object-oriented modelling language – the Unified Modelling Language (UML).

The OMG is not only interested in modelling languages; its primary aim is to enable easy integration of software systems and components using vendor-neutral technologies. The latest step towards this goal is its announcement of the Model-Driven Architecture (MDA) as the basis for future OMG standards.

One of the keys aspects of the MDA is the separation of platform-independent models and platform-specific models. A platform-independent model can be mapped (or translated) to any number of different platform-specific models. Additionally the MDA recognises other separations between models of the same system, from different viewpoints and at different levels of abstraction.

Many problems within software engineering make use of models and translations, and as stated by the authors of [Blaha_Premerlani_96]:

> "Models allow a developer to focus on the essential aspects of an application and defer details. Transformations extend the power of models, as the developer can substitute refinement and optimisation of models for tedious manipulation of code."

UML is the industry standard language for modelling. UML class diagrams have become a standard way of defining the structural aspects of conceptual models. However, little investigation has been done on using UML to specify relationships between such structures. If UML could be used for this purpose, then it would not be necessary to learn a different language to define those relationships.

The aim of this thesis is to investigate the possibilities of using UML to specify relationships (e.g. translations) between models. Specifically, we will focus on Object-Oriented (OO) models, which can be expressed in terms of UML classes, associations and well-formedness constraints. This includes meta-models, which are essentially OO models of the abstract or concrete syntax of languages.

The remainder of this introduction is structured as follows. Section 1.1 discusses two types of relationship that can be specified between models – Translation and Inter-Consistency. Section 1.2 illustrates a number of example applications that make use of such relationships. Section 1.3 states the two primary objectives of this thesis and section 1.4 concludes with an overview of the other chapters.

1.1 Translation and Inter-Consistency

The relationship between two distinct models can be specified as a translation, i.e. the specification of how to create one model from the information provided by the other. In the case of the MDA, the relationship from platform-independent model to platform-specific model is a translation.

Alternatively, the relationship can be more of a peer-to-peer mapping (i.e. a function and its inverse). Neither model is generated from the other, but the models are specifying the same system from different perspectives, possibly containing overlapping information that must be consistent.

From a declarative perspective, the specifications of both of these types of relationship are the same. The relationship defines a mapping between components from one model and components from the other, the difference is in the actions performed as a result of the declarations being invalid.

For a translator, if the mappings specified are invalid, the actions must indicate how to create components of the target model such that the mapping becomes valid. For an inter-viewpoint relationship, the actions may simply flag up an inconsistency, or may attempt to change one or other model to revalidate the mapping specifications.

Models and the relationships between them (whether translations or peer-to-peer mappings) pervade many different types of software system and have been in use for many years, although not always specifically referred to as translations.

There is much work in various domains to which translations are applied. Applications such as compilers, interpreters and pretty printers all perform a translation and the tools for generating them have been called translator-writing systems. Modern Integrated Development Environments (IDEs) and Software Engineering Environments (SEEs) provide multiple views of the same specification, sometimes using the same languages and notations and often using a variety of different ones. There are also complete research fields dedicated to viewpoint specification and their inter and intra consistency.

The questions we ask in this thesis are “Can UML be used to specify translations, or mappings between models, such that the specification of these systems is possible within a single linguistic framework?” Secondly, “Are those specifications a suitable basis for the provision of an implementation?”

The next subsection introduces three application areas whose specifications could be described as model translations.

1.2 Example Applications

Model and translation based specifications are suited to a number of different software systems. Some contain distinct translations, such as performance model generation, where as others such as multi-view systems are more suitably described as containing mappings between different models. Additionally there are those types of system that can be seen as either a mapping or a translation depending on the style of implementation. Compilers, whether for textual or visual languages, relate concrete and abstract syntax models. A batch, single step style of implementation is easily seen as a translation from concrete syntax to abstract syntax to target syntax. Whereas, an

incremental compiler attempts to map concrete components onto their abstract counterparts, performing updates (incrementally) as changes are made.

The following sub-sections discuss a number of possible applications suited to this style of specification.

1.2.1 Automatic Performance Model generation

The model translation technique can be applied to the area of Performance Modelling. This directly illustrates the requirement to convert one model of a system into an alternative model that represents the system using a different set of concepts.

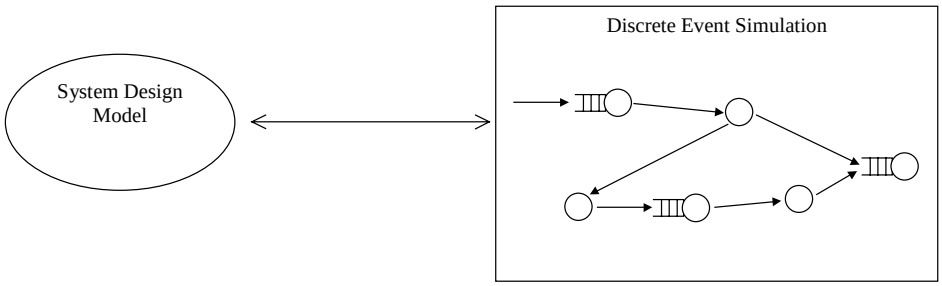

Figure 1 – Translation into a Simulation Model

Figure 1 illustrates a translation from a design model of a system into a Discrete Event Simulation (DES) model of the same system. The translation is specified from design model to DES in order to generate the simulation model and in the reverse so that the results of the simulation can be seen in the context of the original model.

Both models represent the same system but use different concepts to construct their representations. The abstract model could be defined using UML concepts, whereas the DES is defined using queued servers, delays and transactions.

A project (Permabase) that investigated the implementation of this type of system is discussed in Chapter 3.

1.2.2 Visual Language Specification and Editor Implementation

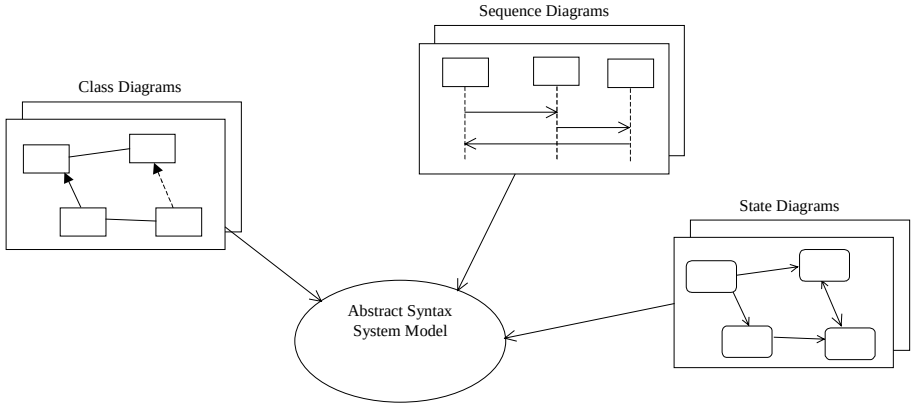

Figure 2 – Multiple UML expressions describing a single system

The technique can be applied within the domain of Visual Languages to give a technique that allows multiple syntactic expressions to define concurrently a single abstract model of the information described ([Akehurst_00]).

This technique is illustrated within Figure 2 with respect to supporting modelling languages such as the Unified Modelling Language (UML). UML embodies a number

of different diagrams types (and syntaxes), which are used to produce multiple diagrams that collectively describe a single system.

1.2.3 Multi User/View Repository Systems

A third possible application of this technique is within the domain of multi user and multi view systems. Such systems often make use of a central repository to store and co-ordinate the information entered by each of the users (or views). It can be a problem, in such systems, to enable each user to have a consistent and up to date view of the information entered and to ensure consistency between multiple views. This type of system is discussed in [Marlin_96], who describes a distributed architecture for supporting multi-view systems using a canonical (central) representation of all views, and uses a broadcast mechanism to propagate changes to each view.

The continuous or active translation approach described by this thesis enables the central repository and each of the user's individual view models of the central information, to be continuously updated. A similar mechanism is used, although it is described as use of an 'Observer Pattern', rather than as a broadcast.

1.2.4 Compilation

The translation technique could also be applied to the task of compiler generation. A program compiler could be built using this technique. Such a compiler would translate between the source code of a program and the destination target language, either an intermediate assembly code or the machine specific instructions.

This style of compilation would give immediate feed back on the validity of the syntax and would not require a separate compilation step. The program would always be executable (assuming the syntax was valid) without requiring compilation after each change. This would give a compiled language a similar appearance to an interpreted one.

Having illustrated four types of system to which we could apply a model and translator based style of specification; the next section highlights the main objectives of the thesis.

1.3 Objectives

The objectives of this thesis are:

1) **To investigate the feasibility of using UML as a basis for a technique that can be used to specify a declarative translation or mapping relationship between two distinct object-oriented models.**

2) **To illustrate whether or not this style of specification can be used to provide an 'active' translator implementation using an approach that can be, at least partially, automated.**

1.4 Thesis Overview

To achieve these objectives the thesis follows the following format:

Chapter 2 Background: provides more detail surrounding the MDA; discusses other work related to the topic of model translation; introduces object-

oriented modelling and the Unified Modelling Language (UML); discusses Graph Grammars and Graph Transformation; and introduces the concept of Patterns, describing in particular the Patterns used in this thesis.

Chapter 3 Permabase: describes the objectives, tasks, results and problems involved in the Permabase project. This chapter illustrates the background that led to the requirement for solutions to the problems addressed by this thesis.

Chapter 4 Translator Specification: defines a UML and OCL based declarative technique for specifying translations or mappings between models.

Chapter 5 Translator Implementation: discusses two possible methods for implementing a model translator given a specification of the form described in the previous chapter.

Chapter 6 Automation: discusses an approach for automatically generating a translator implementation from the UML/OCL based specifications.

Chapter 7 Evaluation: discusses the UML/OCL specification technique and the suggested methods of implementation. The specification technique is evaluated in the context of Cognitive Dimensions and the implementation approaches are compared. Discussion surrounding the use of the techniques on various examples is also included.

Chapter 8 Conclusion: highlights the contributions of the work presented in this thesis showing how the UML/OCL specification technique and the implementation approaches meet the objectives outlined in Chapter 1. This section also proposes some future research that could lead on from the results of the work presented here.

Chapter 2

Background

This chapter starts by discussing the OMG's Model Driven Architecture (MDA) indicating how the work in this thesis relates to their framework. The chapter also describes other work related to the area of model transformation, both directly, and indirectly discussing topics where model translation and mapping is applied. Also included is an overview of topics that aid the understanding of the research presented in the following chapters.

Section 2.1 discusses the OMG's MDA illustrating how the work in this thesis is timely and ideally suited for use within that framework. Section 2.2 looks at work specifically related to model translation. Section 2.3 discusses compilation as a translation technique. Section 2.4 discusses the concept of viewpoints and how model translation is applicable to supporting viewpoint consistency and integration. Section 2.5 presents a brief introduction to the language and notations of UML and OCL. Section 2.6 introduces Graph Grammars and Graph Transformations. Section 2.7 finishes the chapter with a description of some modelling Patterns used as part of the proposed implementation technique for model translators.

2.1 Model Driven Architecture (MDA)

MDA is the OMG's latest initiative towards providing a framework that enables vendor independent and future-proof software-system integration. Starting with CORBA, the OMG has been responsible for the standardisation of a number of different (object-based) technologies. The first of these, CORBA [OMG_00oct] is a standard for supporting interoperability across multiple middleware platforms. UML [OMG_99jun] is the OMG's standard modelling language and is widely adopted across the industrial software engineering community. XMI [OMG_98oct] has been developed as a standard for enabling communication of UML models and CWM [OMG_00jan] as a standard technology for data warehousing.

MDA is an initiative that addresses integration and inter-operability across the software-system life cycle, from initial modelling through design, implementation, management and evolution. Documents such as [Dsouza_01mar] and [OMG_01feb] give good accounts regarding the details of the initiative. The primary relevance of the MDA work in relation to the content of this thesis is its focus on models and the transformations and mappings between them.

A number of concepts, key to the MDA approach, are defined in [OMG_01feb]. These definitions are included here to aid understanding of the following overview of the MDA.

Model A model is a formal specification of the function, structure and/or behaviour of a system.

Abstraction Abstraction is the suppression of irrelevant detail.

Viewpoint A viewpoint is a model of a system based on specific abstraction criteria.

Platform A platform encompasses technological and engineering details that are irrelevant to the fundamental functionality of a software component.

Given these definitions, we can state that the purpose of the MDA is to provide a mechanism for writing specifications that are based on a platform-independent model (PIM). These specifications could be at any level of abstraction and from any viewpoint but fundamentally, the MDA enables integration of several such specifications about the same system.

The MDA focuses on the functionality of a system, not on the technologies that are used to implement it. Hence, it provides for specifications that are independent of specific vendors and of the continuously moving target of the "current best implementation technology".

Implementation of a model is supported by the generation of a platform-specific model (PSM). A platform-specific model is created by transforming the platform-independent model into a model based on the fundamental concepts of the target platform.

UML is proposed as the ideal means for specifying a PIM and [OMG_01feb] proposes that UML can be used to construct profiles for each required PSM. Such profiles are already starting to come into existence, for example the UML profile for CORBA [OMG_00feb].

The mappings between the different models are as important as the specifications of the models themselves. However, no standard technique has been proposed as a mechanism for specifying these mappings.

Within documents such as the IDL to Java Language Mapping [OMG_99jun3] or the IDL to C++ Language Mapping [OMG_99jun2], the specification of the mapping details is achieved using a mix of informal text and examples. Other examples of translations specification within the OMG are those relating to XML. The XML Metadata Interchange (XMI) Specification [OMG_98oct] defines how to translate a MOF model [OMG_00mar] into an XML DTD [W3C_98feb] and the technique for describing the translation uses some formality in its use of EBNF [ISO/IEC_96] and OCL.

The informal mechanism adopted for the IDL mapping specifications makes the documents amenable to human interpretation. However, the specifications are not useable as input for the provision of an automatic translator. The MOF to XMI translation is specified more formally; however the introduction of an additional language, EBNF, raises the question "Would the specification of the translations be possible using UML?"

Given that it is the OMG's philosophy to be consistent in using UML to express the definition of models, is it also possible to use UML to express mappings between

models? The work described in this thesis investigates this possibility and proposes a possible approach.

The next section of this chapter reviews work that is directly related to the topic of model translation.

2.2 Model Translation

Not much reported work directly refers to itself as model translation. However, there is some and this section discusses some of these bodies of work.

2.2.1 Transformation Rules Based on Meta-Modelling

Lemesle presents [Lemesle_98] a transformation technique based on representing the meta-model of both source and target models in formalism based on sNets, a particular style of semantic network with typed nodes [Bézivin_etal_95].

The approach is sound and very similar in theory to the UML/OCL (see sections 2.5 and 2.5.1) approach presented in this thesis. However, the use of a non-standardised notation limits the extent to which it is likely to be adopted.

Any general transformation specification needs to be defined in terms of the source and target meta-models; Lemesle's technique uses sNets to define the meta-models and a set of textual grammar rules to define the transformations between meta-model components.

2.2.2 A Catalogue of Object Model Transformations

The authors of the paper [Blaha_Premerlani_96] give a succinct overview of the theory of model transformation in the context of object-oriented modelling. They describe three classifications of transformation:

1. *Equivalence Transformation*. There is a unique one-to-one relationship between instances of the source and target object models described. Incidental information (such as association or role names) may be lost, but all of the modelled information is retained.
2. *Lossy (Information-losing) Transformation*. The source model is more constrained than the target model. All instances of the source model can be mapped to target models but not all target models can be mapped to a valid source model. Information is lost as part of the transformation.
3. *Information-gaining Transformation*. The source model is less constrained than the target model. A source model instance may not have a valid target model instance, but all target model instances can be generated from a source model instance. Information is gained as part of the transformation, which must be supplied from another source.

The bulk of the paper is concerned with documenting an extensive set of twenty *primitive transformations* (i.e. transformations that cannot be decomposed into simpler ones). These transformations describe operations that can be carried out on an object-oriented model to produce a different model. The transformations described include operations such as:

- adding or removing a basic component (class, association, attribute, etc.);

- changing the properties of a component (e.g. altering the multiplicity of an association);
- moving a component from one container to another;
- combining or splitting two similar type of component;
- converting one type of component into another (e.g. converting an association into a class or vice-versa).

The transformations can be seen as describing the operations a developer would need in order to create a model. They can also be used as the basic building blocks for describing how to convert one model into another, and as such could aid in the identification or description of patterns of inter-model transformations.

2.2.3 ALCHEMIST

ALCHEMIST is described in [Tirri_Lindén_94] as a general-purpose object-oriented transformation generator. It is especially suited to defining mappings between database data-models. Source and target models are described using a textual context-free grammar; the transformation rules are specified using Tree Transformation grammars (TT-grammars, [Keller_etal_84]).

A TT-grammar is a textual specification of a relationship between two syntax trees, each of which is described by a grammar. The specification can be used to perform a transformation in either direction.

The ALCHEMIST toolkit enables the generation of file-to-object, object-to-file and object-to-object translators. Additionally it provides an extensive set of applications for supporting the following tasks: editing source and target grammars; editing the transformation grammar; reusing existing transformations; and managing the transformation specifications.

The approach is limited by its assumption that object models are always tree structures. This may be the case with many database models, but in general, an object model may not follow the structure of a tree. However, the approach provides good evidence that it is possible to implement a general toolkit supporting model translator specifications, and could be useful with respect to the transformation of XML models.

The system could be improved by the addition of graphical notations for specifying the source and target models and the transformation specification. It is the use of textual context-free grammars that limits the technique to tree structures, a move to graph grammars or UML/OCL specifications would remove this limitation.

2.2.4 XSL Transformations (XSLT)

The definitive guide to XSLT is the W3C standard [W3C_99nov], however other sources provide better introductions and tutorials for those wishing to learn the language (e.g. Chapter 14 of the XML Bible, [Harold_99jul]).

XSLT is part of the XSL (Extensible Style Language) set of standards, which are an effort to develop a standard for the presentation of XML (Extensible Modelling Language, [W3C_98feb]). XSL is divided into the formatting part, XSL:FO (Formatting Objects), and the transformation part, XSLT.

A specification in each of these parts can be represented as an XML document. XSLT is, therefore, a language for describing how to transform one XML document into another. XSLT was originally intended for specifying the relationship between a

source XML document and a target XSL:FO specification of how to present that source document. However, as the format specification language is also an XML document, XSLT became a language that can be used for specifying a transformation between any two XML documents.

As XML is intended as a language for communicating and representing data models, XSLT can be applied as a model transformation language. Work discussing this application of XSLT is documented in [Peltier_etal_00].

The work by Peltier et al describes a formalism for expressing transformation rules in a more succinct notation than pure XSLT. They also propose an architecture where models are defined in a higher level notation than XML, e.g. UML or MOF, and transformations are specified in their own notation. These specifications are subsequently mapped to the lower level XML and XSLT representations for executing the transformation.

The use of UML to specify the transformations and a specification of the mapping between that use of UML and XSLT would be more consistent than introducing a new notation. It would be quite feasible to specify such a mapping from the UML/OCL transformation technique specified in this thesis to the XSLT language.

Having looked at bodies of work that directly address the concept of model translation, the next sections discuss larger bodies of work that can be considered as investigating model translation although not explicitly stating so.

2.3 Compilation

A compiler is a tool for performing a translation between two languages. In particular, within the domain of Computer Science, a compiler translates a set of instructions from a higher level language into a lower level language[1]. Compiler technology is also used for other processes such as text formatting or interpreting database query expressions. A good introductory text covering the majority of aspects is [Aho_etal_86].

This section introduces the process of compilation from the perspective of looking at it as a translation. The first sub-section discusses the overall architecture of a compiler, showing how it can be viewed as a cascade of sub-translation processes. The second and third sub-sections discuss in more details the syntax and semantic analysis processes. The final sub-section reflects on compilation as a translation mechanism, in terms of the specification techniques used, and the implementation approaches.

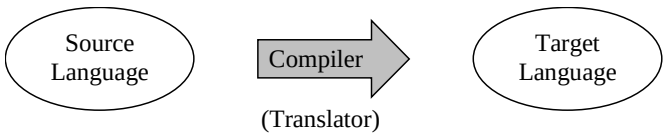

Figure 3 – The Compilation Process

[1] The terms higher and lower are relative to the relationship between the two languages, with lower implying that the language primitives are simpler instructions.

2.3.1 The Compiler Architecture

We can view a compiler as being a translator between a source language and a target language (Figure 3). However, this translation process can be sub-divided into a number of sub-translations, as shown in Figure 4.

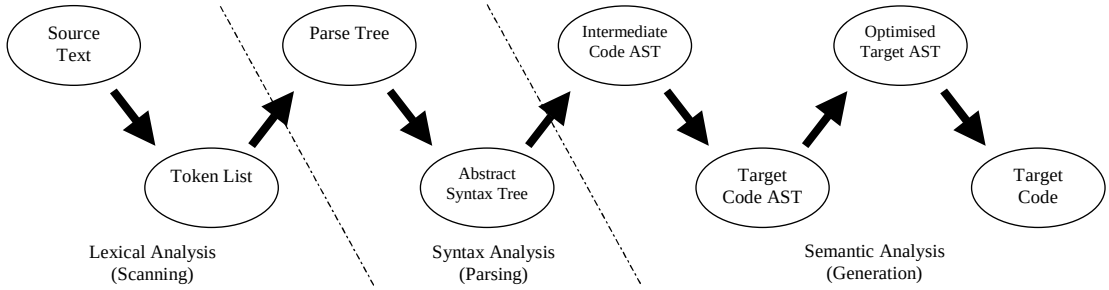

Figure 4 – Phases and Translations in a Compilation Process

Compilation starts with Lexical Analysis. This very simple translation process recognises patterns in sequences of characters and outputs a list of tokens representing each recognised pattern. The patterns are defined by associating a regular expression with a token name that represents the recognised pattern. This process is also known as scanning.

The output of the scanner is *parsed* (Syntax Analysis) to generate a *parse tree* structure based on the source language *phrase grammar*. The phrase grammar specifies rules that group particular tokens into groups (phrases). The parse tree is subsequently converted into a more compact representation, the abstract syntax tree (AST). This step from token list to abstract syntax tree is often implemented as a single translation, skipping the intermediate representation of a parse tree.

Semantic Analysis is the process of interpreting the meaning of a source language expression; the abstract syntax tree is the prime source of information used to carry out the analysis. To analyse the semantics of an expression, it is necessary to have some information regarding the semantic rules of the language. Semantic rules are defined by associating attributes to nodes in the parse tree (or AST), and giving values to the attributes using functions over other attributes. If none of these functions have any side effects, the grammar (and semantic rules) can be known as an attribute grammar.

More usually, the functions in semantic rules do have side effects; such actions are used to generate output for sentences parsed by the grammar. In this situation, we have a translator; the input source expression is scanned, parsed, and analysed to produce output. If the output is in the form of text, then we have a simple, String-to-String, translator.

Alternatively, the output can be a different data structure (or structures) that can be further processed, constructing translators that are more complex. For instance, the output can be a model (AST) of the target code of the compiler, which in turn undergoes optimising transformations before the output target sentence is produced by the compiler.

The next two sub-sections look in more detail at the syntax analysis and semantic analysis phases of compilation.

2.3.2 Parsing

This sub-section discusses the syntax analysis (parsing) process, both as a batch process and as an incremental process. It looks at both the standard technique employed to specify a parser and at the variations in implementation approach.

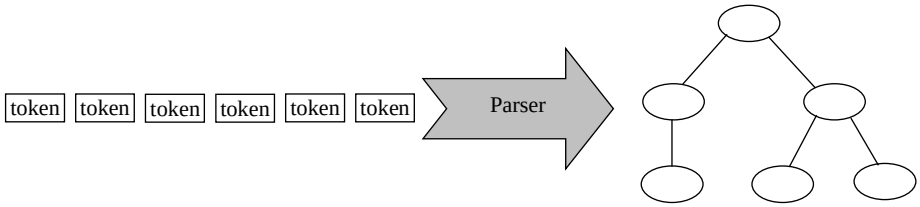

Figure 5 – The Parsing Process

Parsing is a process of translating a sequence of tokens into a tree like data structure (Figure 5). The characteristics of and manner in which the tree is constructed are defined using a grammar and particular parsing technique (LL, LR, LALR [DeRemer_74:1]).

Grammars are defined by a set of terminal symbols, a set of non-terminal symbols, a starting non-terminal and a set of production rules. The terminal symbols match the tokens output from the lexical analysis process and form the primitives used to construct a sentence. The production rules define associations between groups of symbols (terminal and non-terminal), these rules define how the primitive tokens may be combined to form valid sentences for that particular grammar.

The specification of grammars using this technique has a well-defined mathematical foundation [DeRemer_74:1]. Using set theory, a grammar G is defined as a 4-tuple:

G = (N, T, S, P)
N = { non terminal symbols }
T = { terminal symbols }
S = start symbol
P = { production pairs, (α,β) written $\alpha \rightarrow \beta$ |
$\alpha \in V^*$
$\beta \in V^*$
V = N U T }
(V^* is the set of all strings formed from V, including the empty or null string ε)

A *derivation* is a sequence of strings $\alpha_1 ... \alpha_n$ (for n>0) such that there are productions:

$\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_2 ... \rightarrow \alpha_n$ (often denoted as $\alpha_1 \Rightarrow \alpha_n$)

Any string η derivable from S (such that there is a derivation $S \Rightarrow \eta$) is called a *sentential form*. If η consists only of terminal symbols then it is called a sentence of the language generated by the grammar L(G). The language L(G) is the set of all sentences that can be generated by the grammar G, i.e.:

$L(G) = \{ \eta \in T^* \mid S \Rightarrow \eta \}$

Grammar specifications can be classified based on the complexity of the languages they define and the patterns of symbols present in the left and right hand sides of the production rules, this is known as the Chomsky Hierarchy [DeRemer_74:1]. The majority of programming languages fall into the classification of *context-free*, and the parsing algorithms discussed in this sub-section all operate over this class of language.

Grammars are also classified based on the applicability of parsing techniques. Parsing algorithms are based on the direction of scanning the input, order in which a

derivation is built up, and the number of *look-ahead* tokens necessary to distinguish between productions. The scanning direction is either: left to right (L); or right to left (R). The construction of the derivation sequence can be either: top-down or bottom-up. Top-down signifies that the starting point is the start symbol and the derivation is built up moving left to right (L) along a derivation sequence as written above. Bottom-up, signifies that the technique starts with the terminal symbols and builds the derivation sequence in reverse, i.e. right to left (R) as written above. The number of look-ahead tokens is simply indicated using an integer.

By putting these together, we end up with a notation for indicating the classification. The classifications for the most typically used parsing techniques are as follows:

- LR(*k*) – left-to-right, bottom-up, k look-ahead tokens;
- LL(*k*) – left-to-right, top-down, k look-ahead tokens.
- LR(1) – left-to-right, bottom-up, 1 look-ahead token;
- LL(1) – left-to-right, top-down, 1 look-ahead token;
- LR(0) – left-to-right, bottom-up, no look-ahead;

Further more LR(1) grammars can be sub classified into SLR (simple LR) and LALR (look ahead LR) grammars. More information regarding these parsing techniques can be found in [Aho_etal_86].

The above definition of a grammar and an appropriate parsing technique enables syntactic analysis of an input sentence. Parsing is the process of determining a derivation that enables the parsed sentence to be generated from the starting symbol; if the parse is successful, the sentence is syntactically valid. Errors in the syntax of a sentence can be indicated and in some cases, possible corrections can be suggested.

Both the lexical grammar and the parser (phrase structure) grammar can be simply defined using respectively a regular expression language and a grammar language (such as Backus-Naur Form, BNF, or Extended-BNF [ISO/IEC_96]).

The process of lexical analysis is easily modelled using a state machine, and hence automatic construction of a scanner from the input description is possible. The same is true for some classes of phrase structure grammars and specific parsing techniques.

Programs that perform this automatic construction take as input the description of a grammar and output a program that can be used as a parser for the input grammar. Such programs are called Compiler-Compilers or Translator Writing Systems. Particularly well known are the ‘lex’ and ‘yacc’ tools [Levine_etal_92] and their successors such as Flex and Bison [Donnelly_Stallman_00], or one of many other variants.

Irrespective of the particular parsing technique employed, the parsing process is one of identifying a particular sequence of production rules (called a derivation) that can be applied to construct the input sentence. The process of identification was initially seen as a single batch process, requiring the whole sentence to be parsed in order to create the output parse tree. Changes to the input sentence required re-parsing the entire sentence.

Incremental parsers retain the structures produced during the parsing process and reuse them to update the output when changes are made to the input. Primarily this enables reuse of unchanged parts of the parse tree, consequently speeding up the production of the output after edits on the input.

Work by Ghezzi and Mandrioli ([Ghezzi_Mandrioli_79], [Ghezzi_Mandrioli_80]) used the notion of threaded parse trees to store information about the state of a parse. Subsequent changes to the input text are *state matched* to specific points in the parsing process, indicating branches of the parse tree that can be re-used, and those that need to be replaced.

Ghezzi and Mandrioli's algorithm is a bottom-up LR(0) approach to parsing, an alternative approach using *state matching* is proposed by Jalili and Gallier [Jalili_Gallier_82] and can handle LR(1) grammars. There are also other approaches such as [Celentano_78], which is more space efficient, or [Wegman_80] achieving maximal node reuse. These works have been built on in texts such as [Larchevêque_95] which extends Ghezzi and Mandrioli's approach to LR(*k*) grammars, and [Yeh_Kastens_88] which presents termination conditions for incremental parsing enhanced by a skipping heuristic introduced by Wegman.

All of the approaches have different features, being suitable for different classes of grammar, with different space and complexity measures and impose varying constraints on the types of edit that can be performed. Modern work, such as [Wagner_Graham_98], defines a more flexible approach and supports a larger class of grammars. Their approach also uses a parse tree, but, rather than recording and matching the state of the parser at each node, they use a more powerful technique called *sentential-form parsing*. This approach enables the parser to match a non-terminal to an already built segment of parse tree, and re-uses it rather than re-building it.

Parsing produces a parse tree, or AST, and checks that the syntax of the sentence is correct. The next step is to interpret the meaning (semantics) of the parsed sentence; we discuss this process in the next sub-section.

2.3.3 Semantic Analysis

Attribute Grammars [Knuth_68] were initially introduced by Knuth for the purpose of semantic analysis. They extend a standard grammar by attaching attributes to the symbols of the grammar (terminal or non-terminal). Each grammar production is associated with a set of semantic equations, which define the value of the attributes in terms of functions applied to other attributes. There are two classes of attribute, synthesised and inherited. The difference is most easily understood by associating the symbols with their nodes in the parse tree; a synthesised attributes is calculated from attributes of sub-nodes and inherited attribute are calculated from attributes of parent or sibling nodes. The attributes of terminal symbols are often assigned values by the lexical analysis process.

In addition to assigning values to attributes, a semantic function can be defined to perform some kind of action, for example an output action. Thus, as a sentence is parsed, not only are the values of symbol attributes updated, but also output can be produced. For simple compilers, the output actions can drop the generated output of the compiler; however, situations that are more complex require a secondary data structure to be created.

The output actions can be defined such that they build an alternative data structure, for example a model representing the abstract syntax of the target language of the compiler. In this case the execution of the actions performs a Tree-to-Tree

transformation and the grammar (including the semantic actions) can be called a transformational grammar [DeRemer_74:3].

Further work with Attribute Grammars has identified Ordered Attributed Grammars [Kastens_80]. These are a subclass of Attribute Grammars, which enable automatic construction of compiler algorithms.

Often, parsing and semantic analysis are mixed up in a single execution process; the parse tree or AST is not explicitly built. Rather, as each grammar symbol is matched the semantic functions are executed.

The use of an incremental parser raises problems with this mixing of semantic actions into the parsing process. Edits to the source text causing changes in the parse tree will imply different values for symbol attributes, and possibly require the output to be different. If the output actions have already been executed, any change to the parsing order may change the order in which the output actions would have been executed if the sentence had been parsed as a whole. Consequently, an incremental semantic analyser must be used instead.

One approach, described in [Reps_etal_83] is to define everything using attributes, i.e. no actions, and to re-evaluate the affected attributes when changes are made. An alternative, semantic-action, approach requires a designer to specify actions for producing output and additional actions for retracting or undoing the effects of an output action. Generators such as those documented in [Krafft_81] and [Medina-Mora_82] use this approach and an alternative version using a hybrid of the two approaches is proposed in [Johnson_Fischer_82].

To conclude the discussion on compilation, the next sub-section looks at the compilation techniques with respect to the model driven approach and requirements as discussed in Chapter 1.

2.3.4 Analysis / Reflection

From the perspective of compilation, a translator is seen as a generative process as opposed to a relationship between two distinct models. This is particularly apparent with respect to the specification mechanism, a grammar. A grammar describes how to generate a set of sentences. Consequently, any specification based on a grammar will naturally have a generative flavour.

Both lexical parsers based on regular expression grammars and the more complex syntax parsers use a grammar specification in reverse. They detect whether a particular sentence could have been generated from the specified grammar and in the case of syntax analysis, they determine how the sentence could have been produced in terms of the defined grammar rules (productions). The most common technique used to perform this process is by constructing a pattern matching state machine in accordance with the rules of the grammar. Actions associated with the transitions between states create the output of the translation system.

Over time, the implementation techniques for these kinds of translator have become very efficient, even with respect to the more complicated incremental implementations. Automatic compiler construction tools have been built, which significantly ease the process of building a compiler. Eli [Gray_etal_92] is one such compiler construction tool, based on the extensions to Attribute Grammars described in [Kastens_Waite_94]. It takes as input a number of specifications defining the

source language, translation process and output machine code and generates a complete compiler as output.

In accordance with the objectives of this thesis, we try to show that it is possible to use UML and OCL to define such translation processes. This is as an alternative to the standard approach to specifying and implementing the translators based on grammar specifications.

A model driven architecture for the scanning and parsing segments of a compiler could be as shown in Figure 6. The original text, the sequence of tokens and the abstract syntax can all be seen as separate models, with a translation between them.

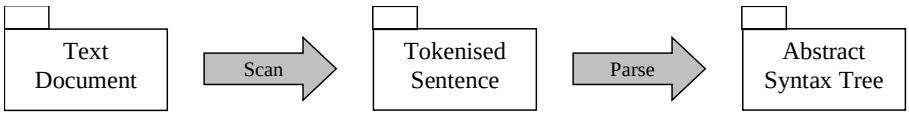

Figure 6 – Compilation as Models and Translators

The specification of the text document is simply a sequence of characters. In accordance with the specification of a lexical analysis grammar, the Tokenised Sentence model must define a number of token types, each matching a specific pattern as defined by a regular expression. The Abstract Syntax Tree (AST) model must (obviously) define the components of the abstract syntax of the language being parsed, and the relationships between them. The translators, 'Scan' and 'Parse' map text strings to tokens and groups of tokens to parts of the AST.

The implementation of systems specified in this manner could be achieved as batch process or as incremental (or active) translations. The choice is a question of mapping the platform-independent specification to a particular platform-specific programming pattern and set of libraries.

A current approach to implementing parsers is to use a state machine, for both incremental and batch based implementations. The incremental approaches save state information in the context of particular input tokens; changes to those tokens can be subsequently parsed by starting the state machine analysis in the appropriate related states.

One could use the proposed constraint based approach in this thesis to specify a parser and take a validation / action approach to implementing one. This approach treats a parser in a similar way to a mapping between two different views as discussed in the next section.

2.4 Multiple Viewpoint Environments

The OMG MDA initiative presents viewpoints as a possible means to divide the specification of a system into a number of separate model specifications. Although separate models, each of the specifications defines details about the same system and hence, there is a requirement to show that they are not inconsistent.

The issue of viewing a specification from multiple viewpoints has been under investigation for several years and there is a large body of research discussing the related problems and their solutions. The next sub-section (2.4.1) traces a path from the domain of compilation (discusses previously) through to the research areas discussing multiple viewpoint specification. The succeeding sub-section (2.4.2)

describes some of that research and 2.4.3 illustrates how the ideas presented in this thesis can be used within the field.

2.4.1 From Compilers to Multiple Viewpoint Environments

Over the last 30 years, tool support for programming and software engineering has evolved from structured editors, to software development environments (SDEs), to the current state of the art in multi-view development environments (MVDEs).

As software-systems have become larger and the languages for building them more complex, tools have been built to aid the software engineer prior to receiving error messages returned by a compiler. In addition, other tools have been built that automate or aid the construction of software development tools, based on the specification of the language in use.

Starting in the 1970's, technologies emerged for constructing structured (or syntax-directed) editors for text based programming languages, such as the Cornell Program Synthesizer [Teitelbaum_Reps_81], MENTOR [Donzeau-Gouge_etal_84] and GNOME [Garlan_Miller_84]. These tools aim to give assistance to a programmer by helping them to write the correct syntax for the language. Initial tools imposed restrictions, limiting the types of possible editing actions. Probably due to these limitations, structured editors never caught on as a general use tool; even so, the associated technology continued to be explored. More advanced systems, such as PECAN [Reiss_85mar], provide multiple views on the code including, in addition to a standard text view, a flow graph view or a view of the incrementally built symbol table.

Along side the development of tools to aid the programming process in a particular language, is the development of technologies to automatically generate those tools for application to any language. These extend the functionality provided by a compiler-compiler, to automatically create structured editors for the input language in addition to the compiler.

These technologies evolved, to provide a complete suit of tools for aiding a programmer. Tools such as The Synthesizer Generator [Reps_Teitelbaum_84] [Reps_Teitelbaum_88] (evolving from the Cornell Program Synthesizer), CENTAUR [Borras_etal_88] (evolving from MENTOR), GANDALF [Habermann_Notkin_96] and PSG [Bahlke_Snelting_86] generate additional tool support such as static analysers, incremental compilers (see above), browsers, and debuggers.

The CENTAUR system for generating programming environments takes as input the specification of a language (syntax and semantics) and it generates a set of tools for aiding programming in that language – text editor, structured editor, pretty-printer, and interpreter/debugger. Additionally, the generated environment allows a user to use multiple views at one time.

With the introduction and wide spread use of languages supporting a modular architecture, additional support is required in order to manager the overall architecture of the systems under construction. These tools became known as Integrated Development Environments (IDEs) and in addition to textual views of the source code, they start to include views that use graphical languages to express some information.

The PROGRES tool [Schürr_94nov] [Schürr_etal_95] is one such, graph grammar based, tool that uses a mixture of graphical and textual syntax to provide multiple

views on the specifications generated by the tool. Additionally, this tool has been used to define other software engineering tools [Schürr_97] (and see below).

Moving beyond environments focussed on specifically supporting the programming task in software development, we get into Software Engineering Environments (SEEs) or Integrated Project Support Environment (IPSEs) that aid the whole software system development process. These support multiple views on the software system, from different perspectives – the program code is not considered the only important factor.

The IPSEN project [Engels_etal_92] makes use of the PROGRES tool to define highly integrated software engineering environments, using attributed graphs to model object structures, software documents and their relationships. The PROGRES language specifies the environment using a mixture of UML like class diagrams and sets of graph re-writing rules, in addition to textual parts and views of the specification.

Earlier work [Engels_etal_86], related to the IPSEN project, presents a technique for structuring structured editors. The approach is based on Attributed Graphs as a means for data storage and it discusses the different types of changes (user increments) that can be performed on the underlying structure.

In the next sub-section, we look at the issues surrounding multiple viewpoint environments. The sub-section ends with a reflection on how the translation techniques proposed by this thesis could be applied to the specification and implementation of such environments.

2.4.2 Viewpoints

Within any complex, large scale system there are likely to be a number of different agents or interested parties. Each agent will have a different view of or perspective on the system and be interested in a different subset of the total information about a system. Each agents perspective does not totally describe the system, but it should totally describe the system from that viewpoint, i.e. it should define all information relevant to the particular agent.

Research in the field of viewpoints extends beyond the provision of multiple view programming environments; in fact, a large section of the field is not tool-based research at all. Technologies such as the RM-ODP [ISO/IEC_95:1] define a non-prescriptive framework in which to define distributed systems.

The RM-ODP framework consists of five viewpoints – enterprise, information, computational, engineering, and technology – which cover the required perspectives of all parties involved in the design process. There is significant research investigating the inter-consistency of specifications written from each of these viewpoints [Boiten_etal_95], [Bowman_etal_96jan], [Dustzadeh_Najm_97].

In [Meyers_93] the author provides an extensive discussion of different architectures for constructing multiple view environments. He favours a canonical representation, in which each view communicates with a common central model. This is contrary to the approach adopted by the RM-ODP, in which each viewpoint contains a separate model and inter-viewpoint consistency is determined on a pair by pair basis. However, if the central model is taken to be a view in its own right, then there is little difference in the approaches.

Finkelstein et. al. [Finkelstein_etal_92] [Nuseibeh_etal_94] describe a viewpoints framework, giving a general definition of what is required in the definition of a viewpoint. They additionally define the need and mechanisms required for defining inter-viewpoint relationships in a generic fashion. For any two viewpoints, there is a set of rules relating each viewpoint. Interestingly, they specify that the rules reside in one or other of the viewpoints as opposed to being part of a separate specification. They make a clear distinction between checking the validity of a rule and the action taken due to a rule being invalid; this they call inconsistency management. The authors clearly support the notion that it is not always possible to resolve an inconsistency, but it is generally informative to indicate them, so that appropriate action can be taken if necessary.

Work reported in [Emmerich_96], [Emmerich_etal_97] and [Abiteboul_etal_94] discuss a suite of tools for specifying multiple view software engineering environments. Central to the approach is a General Object-Oriented Database developed as part of the GOODSTEP project, and the GOODSTEP Tool Specification Language (GTSL). GTSL is used for specifying static and dynamic properties on the central data model as well as mappings between the central model and external views on it.

The requirements for the GOODSTEP database are set out in [Emmerich_etal_93esec] and [Emmerich_etal_93dexa]. Essentially the requirement is for a database system suitable as a central repository for an SDE or process-centred SDE, as described in [Emmerich_95]. The GOODSTEP database extends an existing object-oriented database management system, O^2 [Bancilhon_etal_92], to have additional functionality. In particular, the system is enhanced so that it fires “triggers” whenever a model changes; in essence making the data-models supported by the system, *observable* (see section 2.7.3).

GSTL is used to define the central model, viewed as an abstract syntax graph, and to define actions that should be performed as a result of changes to that model. Interactions are similarly specified using a condition and sequence of actions. (All the GOODSTEP and GTSL work is based on Emmerich’s work as part of his thesis [Emmerich_95]).

2.4.3 Analysis / Reflection

The specification of multiple viewpoint systems can be seen as the specification of a number of models and the specification of the relationships between those models. In the context of compilation translation is a process; however, with respect to a multi-view system, the relationship between two (viewpoint) models is not so easily described as a translation process.

The relationship is an active relationship between the models. Changes in either model must be reflected in the other model, or flagged as being different; i.e. the information presented in different viewpoints is related via a notion of consistency.

In general, it is possible to provide an abstract model for the information presented by a viewpoint, and it is possible to express the model of this information using the UML. For example, work such as [Linington_99sep], [Steen_Derrick_00] and [Bordbar_etal_01] define UML models of the RM-ODP Enterprise and Computational Viewpoints.

In a multi-view system, specifying the viewpoint models is only part of the specification; it is also necessary to specify the relationship between the models. There is no report of an attempt at specifying the relationship between viewpoints using UML, although work related to the MDA implies that this would be useful.

There are two parts to consider with respect to specifying the relationship between models:

- Detection of an inconsistency.
- The actions to perform when an inconsistency is detected.

Finkelstein's and Emmerich's works discuss this issue, each promoting a distinction to be made between a declarative specification of the relationship and the actions to be performed if the constraints are violated. Work in [Feather_96] also discusses the issue, explicitly stating that violation of consistency must be treated in different ways depending on the system. The ability to ignore inconsistencies under certain conditions is crucial for implementing systems capable of supporting partial consistency.

The approach presented in this thesis enables UML to be used as a means to express the relationship between different views in a multiple view environment. This could be in the context of a multi-view SDE (such as the TogetherJ environment [Together]) providing textual and graphical views on the same body of source code (e.g. see Figure 7). Elements of such a system are used to illustrate the use of the techniques proposed in this thesis.

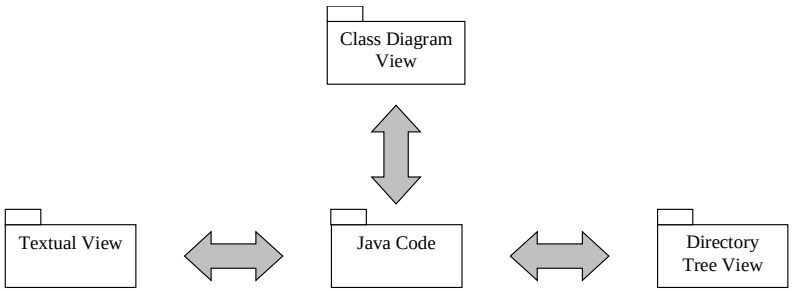

Figure 7 – Model of a Java SDE Using Packages and Translators

Marlin [Marlin_96] presents a similar multi-view architecture based on a canonical representation of the data. Different views are defined, which visualise subsets of the data. The proposed architecture is distributed, with each view a separate process. Changes made in one view are communicated with the central model, which 'broadcasts' the changes to the other views as appropriate. His architecture is similar to that proposed in this thesis and his techniques could be used to extend the approach of this thesis to support a distributed implementation.

Another system using a similar mechanism is the FIELD environment [Reiss_90jun]. This is a system for integrating existing tools into a common SDE. It uses an underlying message passing mechanism for communication between the tools. A tool registers its interest in a particular pattern of message with the message server, which will subsequently forward matching messages to the interested tools.

Both of these approaches, broadcasting and message passing are architecturally similar to the Observer Pattern used as the communication mechanism between

model components in the active implementation approach presented in Chapters 5 and 6.

Alternatively, the technique could be used in a more specification-oriented context, such as to define the relationship between the concepts in the RM-ODP viewpoints. A feature of the proposed technique in such a specification is that it does not specify any action that must be taken if the constraints on the relationship are broken. It just enables the specification of such constraints, supporting the separation between validation checking and actions discussed in [Nuseibeh_etal_94].

OCL is a suitable language for expressing consistency constraints, however, UML does not currently include a suitable action specification language. It is hoped that the action semantics work [OMG_98nov] may provide this in the future. Consequently, the work in this thesis concentrates on the specification of the mapping conditions, leaving the actions to be specified in a platform-specific manner; the specification of actions is left for future work.

2.5 UML

UML stands for Unified Modelling Language; it is the unification of a number of object-oriented notations that existed at during the first half of the 1990's. Originally starting as the combination of the Booch and Object Modelling Technique (OMT) notations, it proceeded to take input from other sources, most notably Jacobson's Object-Oriented Software Engineering (OOSE). When the OMG issued a Request for Proposal (RFP) of a standard object-oriented notation, the UML soon became the focus of an industrial consortium of partners who saw it as the solution. This led to a strong UML 1.0 definition submitted in 1997 as a response to the RFP. Other submitters joined forces with the UML partners and under the management of an OMG working group developed UML 1.1.

The UML consists of a notation and semantics. The notation is the collection of diagrams and the graphical and textual features used within those diagrams, and the semantics defines the meaning of the diagrams and features. The definitive description of the UML is the latest version of the OMG document – [OMG_99jun], however there are a number of books (e.g. [Fowler_Scott], [Booch_etal], [Rumbaugh_etal_99]) which describe the language more informally, though it is important to check which version of the UML they are based on. There is also a tendency for the authors of the books to only include those aspects of the UML with which they have experience, hence some elements of the UML are often left out – when in doubt refer to the OMG document.

There is currently a large amount of work going on in the UML community. The key aspects of this research that are relevant to the work of this thesis is that aimed at improving the definition of the UML semantics. There are papers discussing the problems with the UML and its semantics, e.g. [Breu_etal97], [Akehurst_Waters], [Evans_etal98jun], those that suggest the use of particular semantic models for specific parts, e.g. [Gogolla_Presicce], [Lano_Bicarregui], [Övergaard], and those that address the semantics of UML as a whole or in general such as [Kent_etal].

The research in this thesis draws on some of this UML related semantics work (along with others) and relates it to the problems of interpreting individual UML diagrams and the distinction between concrete syntax, abstract syntax, and semantics. It is hoped that some of the work of this thesis will form part of the background to

contributions proposed to the OMG for version 2.0 of the UML, and in particular relate to the work of the pUML community ([pUML]).

For a good introduction to UML, the references mentioned above are recommended, however for the purposes of this thesis, a brief overview of the various concepts and diagrams is included in Appendix B. Primarily, the work in this thesis makes use of Static Structure (Class) Diagrams and OCL.

2.5.1 The Object Constraint Language (OCL)

The Object Constraint Language was added to UML as of version 1.3. The language is used for specifying constraints about the concepts illustrated within UML diagrams. OCL contains many similar concepts to other formal specification languages, but uses a textual notation that is based solely on the ASCII character set; making it allegedly easier to understand from a software engineer's perspective. A good description of the language and its use can be found in [Warmer_Kleppe], and the definitive OMG issued UML document contains a section on OCL.

The language has a set-based semantics that is very similar to Object Z ([GSmith_99oct]), though the quantifiers and operators are illustrated by using a notation that looks like calling a method on an object in a programming language, rather than by using Greek characters. For example, the following OCL specifies that, for a set of classes, any two different classes must have different names:

SetofClasses->*forAll*(c1, c2 | c1 <> c2 *implies* c1.*name* <> c2.*name*)

The equivalent object Z is shown as follows:

$$\forall c1,c2 \in SetofClasses \bullet c1 \neq c2 \Rightarrow c1.name \neq c2.name$$

Other than this syntactic difference, which often seems unnecessary to those with a formal background, there are other more significant characteristics of the language that determine the way in which it is used within a UML specification.

A constraint is always attached to a model element that gives the constraint a context. For example, the most common usage of OCL constraints is to specify pre and post conditions for an operation, or as invariants for classes in a data model. The operation gives the constraint a context, and hence defines variables that can be used within the constraint – in this case the parameters to the operation and the fields of the class on which the operation is declared.

- There are four basic, predefined, data types: Boolean, Integer, Real, String, which have a number of standard operators defined on them (see the OCL section in [OMG_99jun] for a complete list).
- Any class defined in a UML model of which a constraint is part is a valid type to use within that constraint.
- There are a number of collection types that can be used: Sets, Bags, Sequences, and the generic Collection type.
- The '.' and '->' operators navigate through the class and object structure. The 'x.y' gives a reference to the value of property 'y' of object 'x', the value being, another object, a collection, or a basic data type. The '->' operator is used in an expression such as 's->t', when s is a collection type, and t is a property or operation on that type.

A number of predefined operations are defined for the various collection types. A list of these is included within the OCL section of the UML standard ([OMG_99jun]).

Within this thesis, any OCL statement will show the pre-defined operations and attributes using an italic font.

2.6 Graph Grammars

An object configuration can be thought of as a graph; objects correspond to vertices and links between objects correspond to edges. Hence, it follows that it may be possible to use Graph Theory based techniques, such as Graph Grammars and Graph Transformations to describe models of object configurations and transformations between them. [Bollobás_79] gives a good introduction to graph theory, or see [Wilson_72] for a less formal approach. Appendix C contains an overview of the terms and graph types used in Graph Theory.

String grammars were originally introduced by Chomsky ([Cook_Newson_96]) as a way of creating sentences. Subsequently Graph Grammars ([Claus_etal_78], [Rozenberg_97]) have been developed as a means to describe the creation of a graph (an n-dimensional sentence).

A Graph Grammar is a collection of rules that specify how to create the set of graphs for which the grammar (as a whole) is a valid description. Each rule is a pair, the left hand side (LHS) and right hand side (RHS).

The grammar is interpreted by looking for the pattern of nodes and arcs in the LHS of a rule, if this is found then the graph can be extended as described by the RHS. For example, the grammar shown in Figure 8, is interpreted as described below.

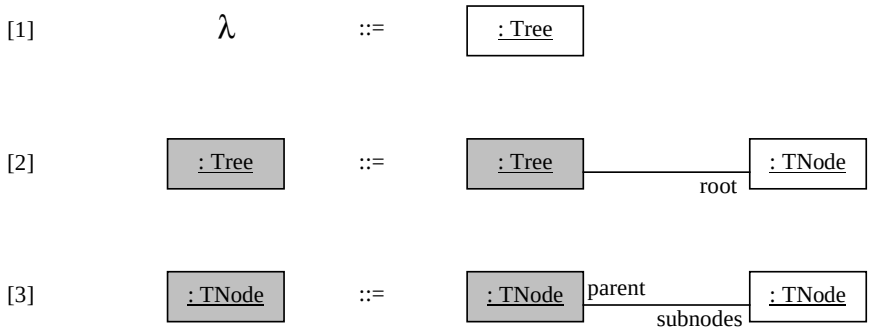

Figure 8 – Graph Grammar for Trees

1. states that a Tree object can be created as a starting point.
2. states that, given a Tree object, a TNode object can be added as its root node.
3. states that, given a TNode object, another TNode object can be added as subnode of the first one.

Thus, a tree-like graph could be built up as illustrated in the sequence of diagrams shown in Figure 9.

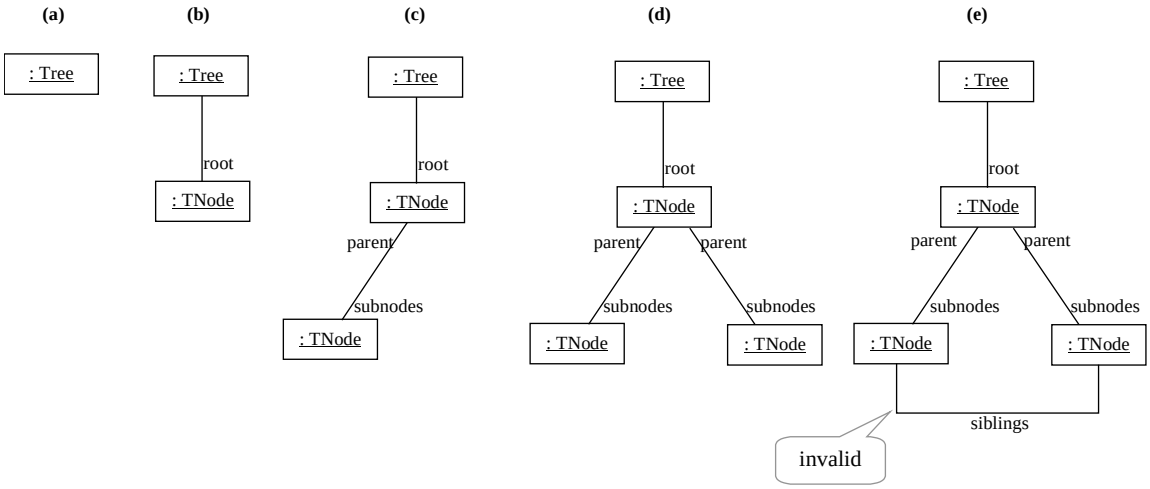

Figure 9 – Creating a Tree

The final diagram (Figure 9e) is not a possible step. This graph cannot be produced from the rules defined above. A change to the graph that connects two TNodes is not a rule that has been defined, the only way to connect TNodes is by creating a new TNode, which is subsequently connected to an existing one. If rule [4] were added as shown in Figure 10, then it would be valid to add connections between two existing TNodes.

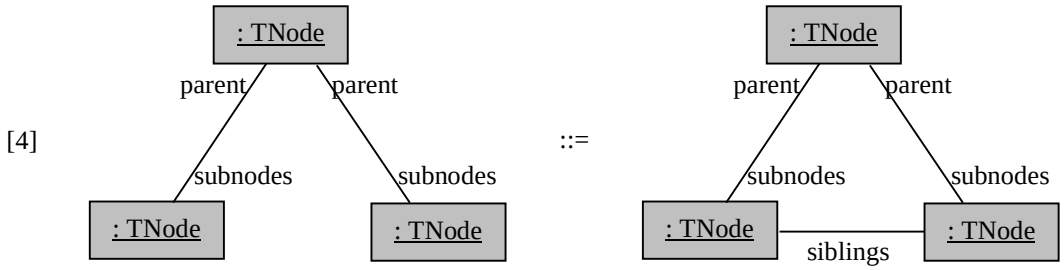

Figure 10 – An Additional Grammar Rule

The notation for illustrating graph grammars is copied from that used in [Rekers_Schürr_96] and [Rekers_Schürr_97]. The shading is an aid (more obviously in Figure 8) to identifying the original LHS components in the RHS subgraphs. The graph built is an object-graph, hence the use also of Object Diagram syntax.

A graph grammar is similar to a string grammar but with the distinction that the rules (or productions) have left and right hand sides which are graphs as opposed to linear groups of strings (tokens).

> "A graph grammar is a system of productions that generates, starting with a distinct axiom (start graph), a certain language of terminal graphs and produces nonterminal graphs as intermediate results."[2]

Graph rewriting (or transformation) systems are closely associated with graph grammars:

> "A graph rewriting system is a set of rules that transforms one instance of a given class of graphs into another instance of the same

[2] [Schürr_94nov], section 1

> class of graphs without distinguishing terminal and nonterminal results."[3]

Graph-rewriting rules alter an existing graph, whereas a graph grammar creates a (or checks an existing) graph according to the defined rules. The term 'graph transformation' can be used synonymously with the term 'graph rewriting'. (A further discussion of Graph Transformation is included in Chapter 4.)

2.7 Patterns

A Pattern, in this context, is a recognised and well-defined solution to a recognised and well-defined problem. They apply to may different domains and in fact, first recognition of them is attributed to Christopher Alexander with respect to the building and architecture industry. He states that:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Alexander_etal_77]

With respect to object-oriented design, the current best introduction to design patterns is the book "Design Patterns, Elements of reusable Object-Oriented Software" ([Gamma_etal_94]). This book is intended as a catalogue of currently recognised design patterns and describes 23 different patterns covering creational, structural and behavioural problems.

This thesis makes use of three of these patterns with respect to providing an implementation of a model translator (Chapter 5); these patterns are described in the following subsections.

2.7.1 Builder Pattern

This creational pattern is used to separate the construction of a complex object from its representation. In essence, it provides a way of creating a complex network of objects without needing to explicitly know how to create those objects.

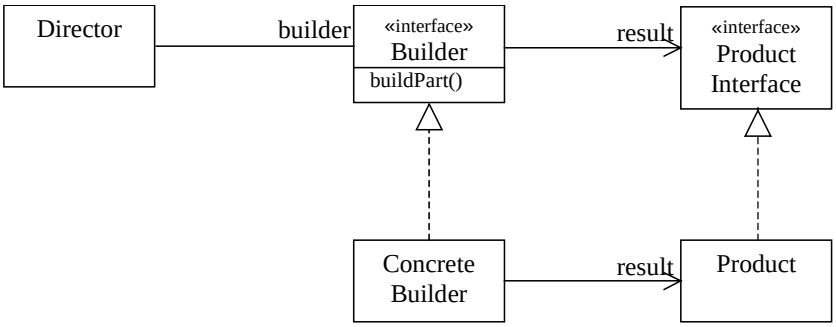

Figure 11 – Builder Pattern Architecture

Figure 11 shows a UML class diagram illustrating the architecture of the participants in a builder pattern. There are four participants in the pattern; these are:

1. The *Director*; is the object causing the complex product to be built; there may be more than one director involved in building the same product.

[3] [Schürr_94nov], section 1

2. The *Builder* interface; is an abstract interface defining the instructions (methods) that can be used to create the product.
3. The *Concrete Builder*; is a specific implementation of the builder interface. It builds and keeps track of a particular implementation of the product.
4. The *Product*; is the complex object or network of objects being built. The product would generally have a set of abstract interfaces that define the product and it's components; these are referenced by the Builder and other users of the product. The Concrete Builder creates a particular implementation of those interfaces.

For the purposes of the implementations described in this chapter, the *Product* is a particular model. This pattern enables the model components to be defined using a set of interfaces; particular implementations of the model also implement a *Concrete Builder* and can hence be constructed without knowledge of the particular model implementation.

2.7.2 Visitor Pattern

The Visitor pattern implements a method of enumerating over every element of an object structure (model). It is used to add operations that are performed on the entire object structure, without altering the classes that define the structure.

The Visitor pattern implements a technique called "double-dispatch"; most OO languages implement a form of single-dispatch. In single-dispatch languages, two criteria determine which operation fulfils a request for a method call – the name of the method and the type of the target for that method. Double-dispatch means that the operation that fulfils the method depends on the method name and the types of *two* target objects.

With respect to the use of the Visitor pattern, an architecture is arranged so that a "visit" method call is fulfilled by an operation depending upon the type of the element to be visited and the type of the object doing the visiting.

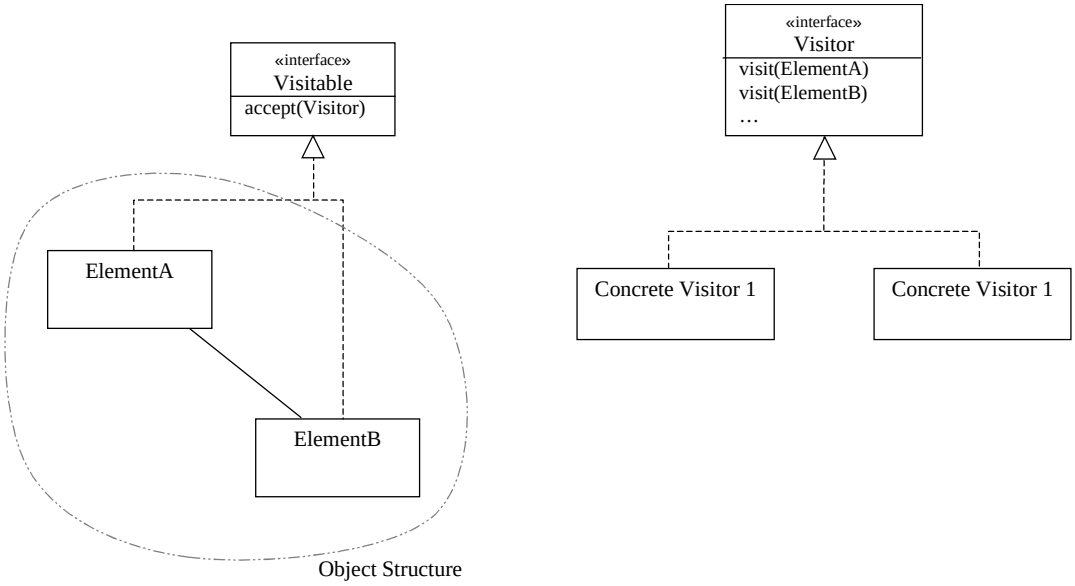

Figure 12 – Architectural Elements of the Visitor Pattern

The class diagram in Figure 12 illustrates the architecture and participants in a Visitor pattern. Each element of the object structure implements the *Visitable* interface. A

Visitor interface for the object structure is defined, containing a visit method for each element and multiple *Concrete Visitor*s can be created that implement this interface.

The Visitor pattern is classified as a behavioural pattern; it is the behaviour defined by the accept method that is the important part of the patterns definition. This method provides the implementation of the double-dispatch behaviour. Figure 13 illustrates this behaviour and it is described below.

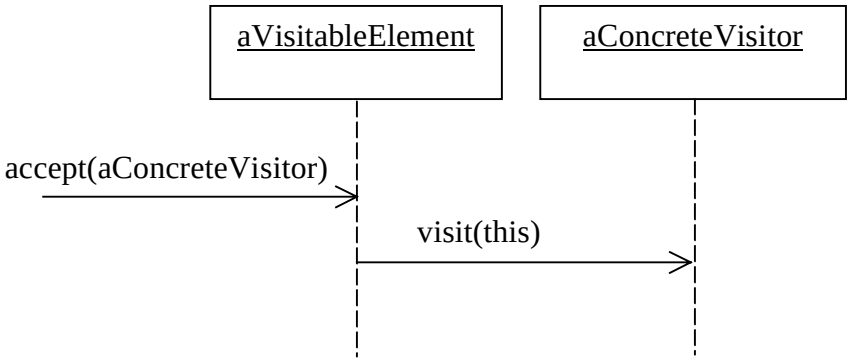

Figure 13 – Behaviour of the "accept" Method

In order for it to be 'visitable', each element in the object structure must implement an "accept" method (defined by the *Visitable* interface). This "accept" implementation calls the visit method for that particular type of element on the particular Visitor that is passed to it.

Any number of *Concrete Visitor* objects can be implemented, defining different behaviour within its visit methods; this gives the mechanism for adding behaviour to the object structure without changing the elements themselves.

The enumeration or traversal path across the object structure can be defined either within the Concrete Visitors or within the implementations of the "accept" methods. Implementing the traversal within the visitors allows different traversal paths to be taken by different visitors. However, often only one path is required, in which case it is more efficient to implement the path within the accept methods reducing the behaviour that must be defined in the visitors.

2.7.3 Observer Pattern

The Observer pattern is a behavioural interaction between objects conforming to two participants – an Observable object and an Observer. Each Observable object may be observed by a number of Observers, but the interaction between Observable and Observer is the same in each case.

The Observer listens to or watches the Observable object and causes some action to occur as a result of changes to the observed (Observable) object. The implementation of this pattern usually takes a form following the sequence of events as illustrated in Figure 14 and described below.

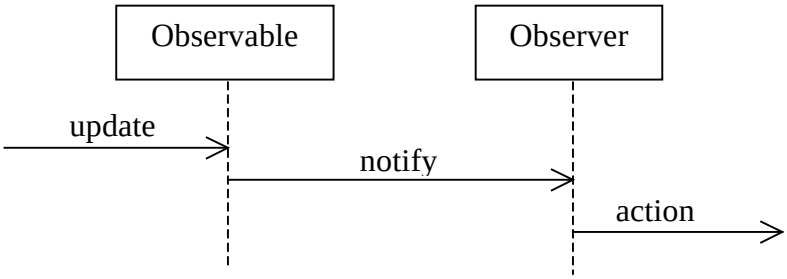

Figure 14 – Behaviour of Observer Pattern

1. the Observable object is updated or changed,
2. it notifies its Observers,
3. the Observers execute the defined action.

Components supporting the implementation of this pattern can take on a number of forms, these are discussed in the following subsection.

2.7.3.1 Observer Pattern Support

Implementation of the Observer pattern is usually supported by an 'Observer' interface and an abstract 'Observable' support class. Figure 15 shows a UML specification of a basic pair of components that support implementation of the pattern.

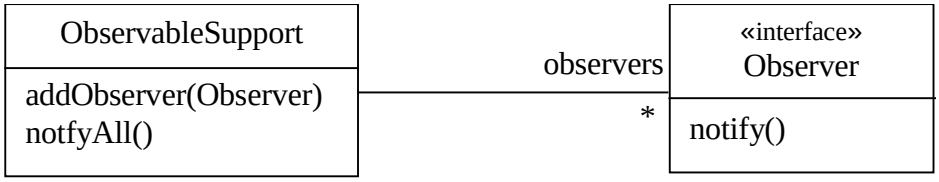

Figure 15 – Observer pattern support definitions

Observable objects extend the ObservableSupport class and observers implement the Observer interface. Observers must register themselves with the Observable object, which 'notifies' all its observers whenever something changes its observable state.

This implementation is acceptable for the simplest of situations. However, often observers require information regarding the nature of the change that has occurred, needing at least to know which object has changed (the source of the notification) and often some description of the change that has occurred. This leads to a more complex implementation of the pattern and it's supporting classes.

One option, support for which is provided in the Java class library 'java.util', is to pass extra information to the observer via the 'notify' method. However, a more flexible method uses the concept of events.

The event notification variation on this pattern often renames Observers as "Listeners". Observable objects "fire" events, which are received by its *Listeners*. The listener may have a number of different 'receive' methods that are used for different event types. An Event object is "fired" (created and passed to the appropriate receive method), which contains information describing the occurrence of that event.

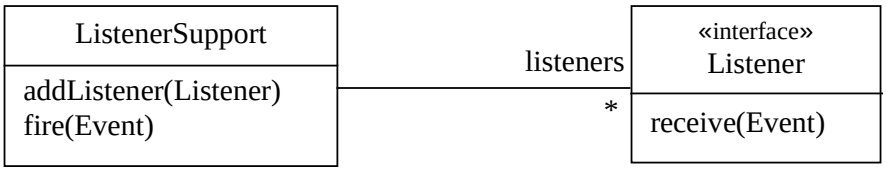

Figure 16 – Event notification pattern support

Figure 16 illustrates the definition of a basic support class and interface for the event notification observer pattern, which includes support for an event object. Other than the names of the classes and methods, it can be seen to be very similar to the basic observer support components shown in Figure 15. Behaviourally the *ListenerSupport* and *Listener* components operate almost identically to the *ObservableSupport* and *Observer* classes; the difference being the passing of an *Event* object to the *Listener*.

This implementation of the observer pattern is used extensively throughout the Java class libraries. The libraries 'java.awt', 'java.beans' and in particular 'javax.swing', all provide various event notification implementations of the observer pattern.

The implementations described in this thesis require the use of the event notification style of observer pattern support. However, none of the sets of components described above provides ideal support for it. The 'java.util' components do not cleanly support use of the event notification mechanism, and the 'listener' based support components do not reflect the names "observer" and "observable" as used by the pattern definition.

Consequently, bespoke implementations have been produced. Within the examples and discussion contained in this thesis, the Observer pattern is supported using the components shown in Figure 17. These are a variation on the observer and event notification components discussed earlier (Figure 15 and Figure 16), but are defined using names that more appropriately indicate their use.

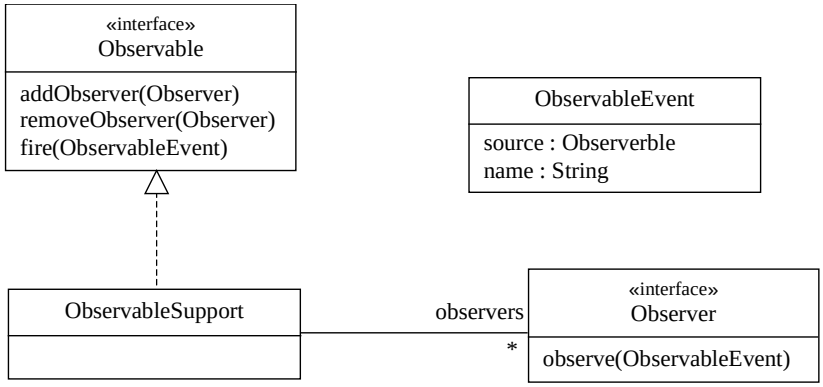

Figure 17 – Observable Event support

The supporting components shown in Figure 17 define both *Observable* and *Observer* interfaces. The *ObservableSupport* class implements the *Observable* interface and the *ObservableEvent* class is used as the event information carrier; it contains two attributes that can be used to describe the event:

- a source – referring to the Observable object that generated the event; and
- a name – that can be used to provide additional information describing the event.

2.8 Summary

This chapter has discussed a variety of topics that, together, form a foundation for the research contained in the rest of this thesis.

The chapter has shown how a UML based specification mechanism for translators is timely and useful with respect to the OMG's Model Driven Architecture initiative and that little existing research directly addresses this issue. However, the idea of translation between language models has a large body of research in the field of compilation and the concept of consistency between multiple viewpoint models also has a significant research background. The chapter has shown how these bodies of research relate to the ideas presented in this thesis.

Additionally, this chapter has introduced the concept of Graph Grammars, which are discussed further in relation to graph transformations, in Chapter 4. It also gives some background and an overview of the UML and related OCL, which are used extensively throughout the thesis. Finally, the chapter describes some modelling patterns that are used in Chapter 5 to describe translator implementation frameworks.

The next chapter introduces the Permabase project. This project illustrates a particular requirement for a translator specification technique and the authors work on this project provided some initial experience in constructing model translators.

Chapter 3

Permabase

A significant portion of the research, in particular the practical experimentation, carried out as part of this thesis has been drawn from the author's work on the Permabase project. This chapter gives an overview and evaluation of the project showing how the project gave rise to the issues addressed by this thesis.

For clarity, Section 3.1.1 provides a distinction between the elements of the project that were specific contributions by the author and those that were joint work between the author and other project members.

3.1 Overview and History

The Permabase project ([Waters_etal], [Utton_Hill], [Utton_Martin]) aimed to provide performance feedback as part of the object-oriented design process for distributed systems. Permabase hides the expertise needed to create a performance model (simulation or analytic) from the system designer, by automatically generating the performance model from the design model. The feedback provided by the performance model allows the designer to change, alter, or confirm, design decisions as necessary throughout the system design process.

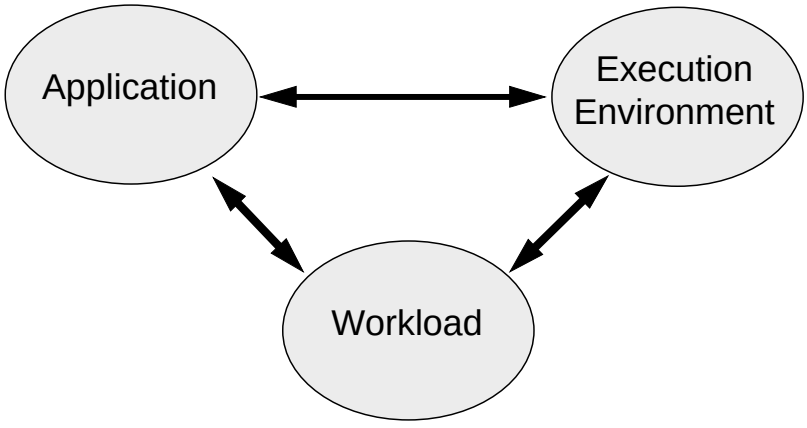

Figure 18 – Permabase domains of interest [Martin_Utton]

The project identified three domains of interest within the specification of a distributed system. Each of these areas must be specified to give sufficient information such that a performance model could be created. These separate domains were originally defined as follows:

Workload Specification – The specification of the "work force" driving the system. These may be human operators, other systems, or simply a model of "miscellaneous" other work being carried out on shared system components.

Application Specification – The specification of the system logic. Primarily this is assumed to be software, but it may be hard-wired (firmware) or hardware logic.

Execution Environment Specification – The specification of the physical environment of processors, networks, and other resources that the system operates over.

The intention was that each of these specifications should be described in an appropriate (standard) notation, and by using a suitable CASE tool for that notation. The details entered into the CASE tools form the source of the specifications to be used by the automatic generation process.

The generation process consists of three stages (as shown in Figure 19):

1. Composition of the information from the three specifications into a "Composite Model Data Structure" (CMDS). This involved the mapping and translation from the representation of the specification in the visualisation input tools in to the concepts defined in the CMDS.
2. Transformations on the CMDS to check for inconsistencies, and to refine the model. Such transformations included transposing the input representation of the system behaviour (in the form of multiple interaction diagrams) into a class-centric representation, more suited to the technique used for performance model generation.
3. Translation from the CMDS into the performance model. This involved the definition of a mapping from the (meta-level) CMDS concepts to the components of a particular performance model engine, and the subsequent algorithm for translating instances of the CMDS (specific design models) into a performance model. One such mapping was to queues, delays and transactions for a discrete event simulation engine, and another to places and transitions for a coloured petri-net style of engine.

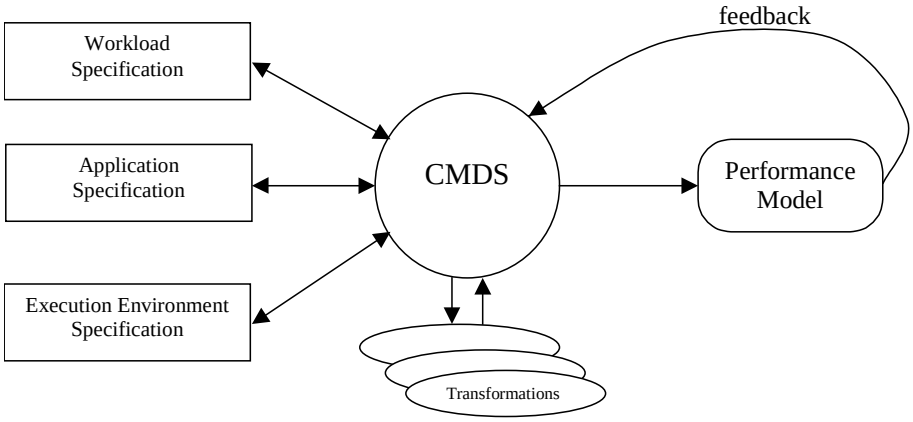

Figure 19 – Initial Permabase Architecture

The decision was taken (based on BT software design practice), to focus on the use of object-oriented designs for the application specification, but there did not appear to be any standard for the specification of the information required in the workload or execution environment specifications. BT practice for object-oriented software design was, when the project started, to use the Booch notation and method (or a slight adaptation of it, [Harwood]), supported by the Rational Rose modelling tool [Rational].

Initially the project looked to this notation for the facility to specify the workload and execution environment information, but the notation did not appear to address

workload specification at all, and physical environment specification was limited[4]. Due to this lack of a standard notation, the project adopted a BT in-house tool that was used for capturing network and system design information. There is no standard notation that the tool supports, it simply allows for the specification of types of node, attributes of the nodes, and connections between the instances of nodes.

As the project evolved it became apparent that these three specifications did not contain all the required information. Although the areas, between them, contained the specification of all the components, it was realised that how the components from each area are connected is of concern. An initial thought was to show in the Execution Environment Specification the connections to the components defined in the other areas. However, this caused the specification to become cluttered and invalidated the definition (see above) of this specification area, as it no longer defined only execution environment information.

The need to specify the interconnections of components from the three specification areas led to the introduction of a fourth specification area – the system scenario specification – which alters the architecture to that shown in Figure 20.

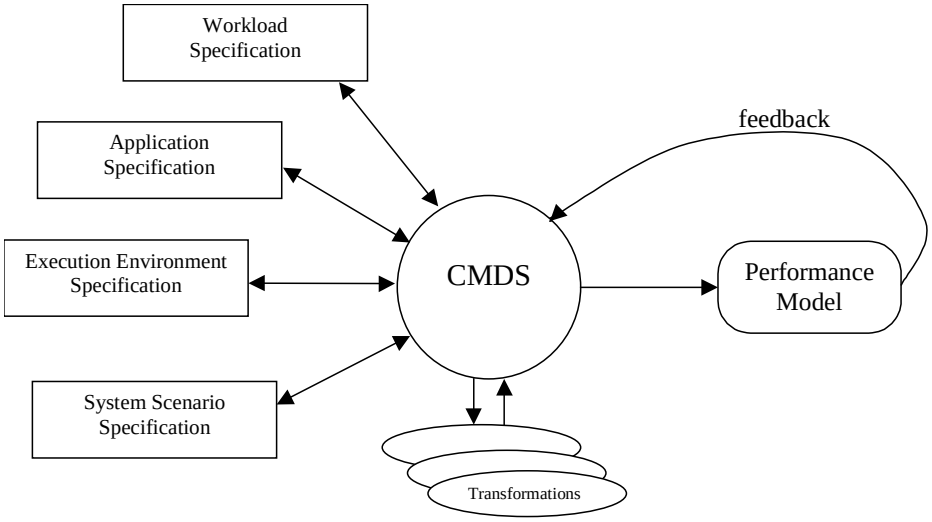

Figure 20 – Permabase Architecture

The existence of the system scenario specification led to the redefinition of the other three specification areas to be *types* of component. This redefinition occurred in part for efficiency, so that there did not need to be repetition of the specification of parts of the system, and in part to draw the whole specification style closer to the object-oriented design pattern of templates and instances. Hence, the four specification areas, required for the definition of a distributed system (for performance analysis purposes), evolved to:

1. Workload Specification – The specification of the *types of* "work force" *component* driving the system.
2. Application Specification – The specification of the *types of* system logic *component*.
3. Execution Environment Specification – The specification of the *types of component* to be used in the physical environment.

[4] Although some of those limits were imposed by the lack of support by the Rational Rose tool, rather than by the notation itself.

4. System Scenario Specification – The specification of a particular (distributed) system instance, in terms of the instances of the types (or classes) of component defined in the other three specifications and how they are connected.

During the lifetime of the project, the UML [OMG_99jun] arose to its position, as the leading (standard) notation for the specification of object-oriented systems. Consequently, it was adopted by Permabase to replace the Booch notation, due to the extra features and improved usability it offered, and because BT adopted it as their standard practice. The use of UML within the project caused a re-assessment of the methods used to describe the source specifications. In particular the extensibility feature of UML opened up the possibility of using it as a standard notation for all (four) specification areas.

In practice, the project used UML for the three 'component type' specification areas – Workload, Application and Execution Environment. The specification of the system scenario was left to the BT in-house tool, as the initial versions of UML did not contain adequate facilities to specify this information. Subsequent revisions to UML, and changes to the way system scenario specifications are defined, enable a mixture of UML class, object, and deployment diagrams to be used; however this was not implemented during the life-time of the Permabase project.

The project ideas were implemented in the form of three prototypes:

1. The first used Booch notation and the Rational Rose tool for input of the Application specification, and bespoke notations supported by the BT in-house tool for the other specifications. The Behaviour of the application components was specified using state machines. The Performance model engine used was a Discrete Event Simulation (DES) engine.
2. The second prototype was based on the UML notation, supported by a new version of Rational Rose. The architecture evolved to that shown in Figure 20 above, and hence we made more use of the UML notation. Feed back from BT based on their use of the first prototype indicated that they would prefer to specify behaviour using Interaction (Sequence or Collaboration) Diagrams, so this became the supported technique. Performance model generation and execution proved to be time consuming using the DES engine, and a faster mechanism was required, particularly for simple designs. Therefore, the second prototype made use of a tool, named RiscSim, built at UKC based on Coloured Petri-nets ([Linington_99apr]), and incorporating some facilities for time and resource usage. This performance engine proved to be much faster for building and executing models; however, it did not provide support for the dynamic binding or dynamic object creation functionality required by some applications.
3. The third prototype used the same input techniques (with minor evolutionary improvements), but added some validation algorithms to be performed over the CMDS. It used an improved version of the DES performance engine, which supported dynamic functionality, and used faster transformation and execution techniques than the first prototype. The DES solution could not match the Petri-net one in terms of speed, but it did provide the functionality required for the main test Case Study.

The final prototype was tested using a Directory Enquiry System case study, specified by BT. The system involved a high work-rate of several thousand calls per hour, processed by a network of over three thousand processing nodes. The performance output of the prototype generated performance model was compared with the output

of another performance model, hand generated by an expert performance-modelling engineer, for validation purposes.

The results of the automatically generated model were statistically equivalent to the hand-generated model. However, the main disadvantage found with the automatically generated model is that it took significantly longer to execute, and hence produce results. This is believed to be a problem common to automatic generation in general and is, in particular, similar to problems encountered within the compiler community. Compiled code used to be renowned for being slower than hand coded assembler, though as can be seen with modern-day compilers the problems have been largely eliminated. It is believed that these techniques could be applied to the automatic performance model generation to aid the improvement of the generated model's execution speed.

3.1.1 Identification of the author's work

This subsection defines which parts of the work carried out within the Permabase project, that are of relevance to this thesis, are solely contributions provided by the author.

The development of the architecture of the Permabase prototype was the result of joint discussion with all members of the project team. The author and one other member (Andrew Symes) were given the task of implementing the prototype based on this architecture. This involved:

- the specification and implementation of the models involved in the system;
- the development of an implementation technique for transforming data from one model into another; and
- implementation of each of the transformations involved in the system.

The authors contributions were as follows:

- specification of how to use UML for specifying systems in such a way that performance models can be generated;
- development of the technique for implementing the translators;
- specification of the CMDS using UML;
- implementation of the CMDS, including
  - development of a set of library components supporting basic persistent storage and searching functionality
  - implementation of an automatic generator for the implemented CMDS data model from its UML specification;
- specification and implementation of the translator from the input, UML, diagrams to the CMDS;
- implementation of the translator from the CMDS to the RiscSim performance engine;

Implementation and specification of the other models and translators was carried out jointly with, or solely by, other project members. This included:

- specification of the translator from the CMDS to the DES engine;
- implementation of the translator from the CMDS to the DES engine;
- specification of the translator from the CMDS to RiscSim;
- specification and implementation of the translator from the BT Configurator tool to the CMDS.

3.2 Analysis of the Permabase Prototype System

The Permabase prototype consists of two types of component, models and translation processes. There are a number of different models within the system and a number of translation processes that convert information from one model into another.

At one side of the system, there are concrete syntax models capturing the input specifications of the system. These are translated into a common composite model – the CMDS.

Within the CMDS, there are different model representations of some parts of the specification. For instance, the behaviour is stored as a set of interaction descriptions (drawn from sequence and collaboration diagram inputs); this is converted by one of the transformations into an alternative 'class centric' model of the behaviour.

Each of the performance modelling engines requires a different type of model as input and the information from the CMDS must be translated into the appropriate model form for input to these engines.

The results produced by the performance engines must be translated back into the CMDS model and subsequently into the original specification models entered by the user in order that they are presented in an understandable context.

Considering only the final version of the Permabase system, it contains:

- Four input specification-models,
- A central repository model,
- Two different performance-engine models, and
- Eight translations between models.

3.2.1 Specifying the Prototype

Given that the system consists of a collection of models and translation processes, the specification of the system is correspondingly a specification of models and translators.

There are many ways to specify a model; the current practice in modelling is to use an object-oriented modelling style and consequently an object-oriented modelling language. The UML is the current standard for object-oriented modelling and consequently this language was chosen as the system specification language.

For specifying the models, UML is perfectly suited; however, the specification of the translation processes or translators was not possible. During the project lifetime, no recognised technique was discovered for specifying translations or relationships between models using the facilities of the UML language.

3.3 Implementing the Prototype

There are two aspects to consider with respect to the implementation of the prototype, the implementation of the models and the implementation of the translators.

3.3.1 Model Implementation

Model implementation was straightforward. A small tool was created for automatically creating the model implementations from a UML model specification.

The tool takes a UML specification as input using the Rational Rose tool and generates a C++ or Java based model implementation.

The generated model implementations were tailored for use with a pre-defined library of components that give some support for basic database functionality such as searching the model and persistent storage.

This enabled fast generation of the model implementations as the models evolved over the duration of the project. Its main use was the generation of successive versions of the CMDS.

3.3.2 Translator Implementation

The other aspect, of implementing the translators, was more complex. Four main translators were implemented over the course of the project. Two translators were used to process the input specifications and create a composite model of the system in the CMDS.

1. Rational Rose → CMDS. A number of variations on the Rose to CMDS translators were built, some were evolutions as either the Rose tool or the CMDS changed version. In the final prototype, three separate translator variations were used to convert and merge the Application, Execution Environment and Workload specifications into the CMDS.
2. Configurator → CMDS. The final prototype makes use of the BT in-house tool called 'Configurator' to input System Scenario specifications, hence a translator was built to convert and merge this information into the CMDS.

The other two translators were used to process a CMDS model and generate a "back-end" performance model. These were the more complex translators to implement as the target domains were composed using a very different structure to the source CMDS model.

1. CMDS → SES. Two versions of the CMDS to SES translator were built. One for the first version of the prototype and one for the final version. These translators were the most complex to implement as SES did not have a simple mechanism for automatically generating input models. The scripting language available was targeted at querying existing models rather than building them from scratch, however, the task was achieved using this language.
2. CMDS → RiscSim. This translator was not difficult to implement, once a satisfactory translation process had been determined. The input is of the form of a text file with a well-defined structure.

Related to both of these last two translators are translators in the reverse direction, which feedback the results generated by the performance engine into the CMDS.

The technique used to implement each of these translators is an approach similar to compilation. The source model is traversed (parsed) and the target model is created during this traversal.

The main difficulty with implementing each of these translators was not being able to create the implementation in accordance with a suitable specification. As stated previously the models were easily specified using UML class diagrams, but there was no suitable technique for specifying the translation process. This is relevant not only from a specification perspective, but also from the point of view of implementation; it

is much easier to implement from a specification than it is to create an implementation directly.

3.4 Evaluation of the Prototype

The final Permabase prototype was a successful proof of concept demonstrator. It proved that it was possible to automatically create a performance model from a system design; this was the aim of the project.

The architecture of the prototype system proved to be a good model on which to base the construction of this type of tool. The architecture involved the use of different models to represent data in the most suitable form for its most immediate processing, be that input of the data (for the source specifications) or execution of the model within the performance engine. These models were subsequently translated into a central data model and out to other forms for alternative processing.

This architecture enabled the addition of new processing techniques (such as alternative specification tools, or additional performance engines) without compromising or needing to modify existing functionality.

However, there were significant disadvantages, both with respect to the use of UML as an input specification language and with the technique used to implement the translations. The following subsections discuss some of the issues.

3.4.1 Translation Processes

The main problem with the translation process is the time it takes to execute a translation. The models can be very large, in the order of tens of thousands of objects, and consequently the translation processes can take many minutes or even hours to finish. In addition, any change to the source model requires the entire translation process to be re-executed; hence building the required target model can be a very time consuming process.

This was not the intention of the project; the generation of performance models is intended to aid the designer giving *quick* feedback regarding the design.

Although the size of the models could be reduced by using different representations of the data and by making more use of repetition attributes[5] the models can still be quite large and translations take a significant time to execute.

The speed of the translation processes is a by-product of the technique used for implementation. A traversal technique was used, this technique required that the target model be built as the process traverses the source model.

This implementation technique focuses on the specification of the source and target models and the translation is a secondary aspect. The specification of the source and target models is possible where as there is no specification of the translation process.

This led to the traversal-based implementation approach, which focuses on implementing the parts of the problem that are specified; however, the approach causes the performance problems outlined above. The performance could be partially improved by marking altered subtrees and only traversing the necessary parts.

[5] An attribute in a data object indicating that it represents many instances of identical objects.

A better implementation technique would be one that focuses on the translations and one that would enable incremental updates to a particular model's translation. To achieve this a specification technique such as that proposed in this thesis would be necessary.

3.4.2 Problems with UML

This subsection describes the requirements of the UML for the purpose of performance model generation. It identifies the key deficiencies of the UML and the interpretation of its meta-model for this purpose and gives a brief account of some of the proposed solutions. These issues, originally based on the specification of UML version 1.1, were presented in [Akehurst_Waters] at the Workshop on Rigorous Modelling and Analysis with the UML.

3.4.2.1 The refinement from design concepts to implementation concepts.

A performance model predicts the performance of an implementation of the designed system. Thus, to interpret a design model in this context, it is necessary to have a precise understanding of how each design concept relates to the implemented system functionality and how it affects the other concepts to which it is related.

This understanding enables the specification of a translation process from design concepts to possible implementation concepts and hence into performance model concepts that will correctly model the behaviour of the designed system.

The initial versions of UML (used within Permabase) were imprecise and ambiguous in the meaning of its concepts in many areas. For example, the relationship between Associations, Attributes, States and Classes; Classes are related to each of these concepts, but how are they related to each other?

From the perspective of an implementation model some of these concepts could be considered notational, but exist as part of the meta-model definition. There is no definition of how they are intended to be refined into implementation concepts. For example, Associations and States can both be implemented as Attributes; within a design or analysis model they can exist as separate concepts, but common implementation languages (C++, Java) do not contain such concepts.

The proposed notion of UML profiles may provide a solution to this. Implementation language profiles could be defined and the refinement relationship between the analysis or design concepts and their implementation counterparts can be specified.

3.4.2.2 Separation and precise relationship between syntax and concepts

The UML meta-model defines a set of modelling concepts, which have a separate description to the UML notation used to visualise the concepts; the UML standard uses a textual description to define the relationship between the two. This can cause confusion or ambiguity when trying to establish a precise interpretation of a diagram in order to extract a model of the expression specified by the diagram.

An alternative approach is to define separately the notational model and meta-model, then precisely define a mapping from one to the other. This would also enable easy specification of alternative syntax representations of the same meta-model concepts, as is currently the case with sequence and collaboration diagrams. The technique presented in this thesis could be used for this purpose.

3.4.2.3 Need for software, hardware and behavioural specifications.

The most mature area of the UML is that used for the specification of the software structure. However, in order to generate a performance model, it is equally important to specify the behaviour of the system and to specify the hardware on which it will execute.

The hardware provides the system resources and the depletion of these is what ultimately limits the performance. The behaviour of the system determines the order and quantity of the resources used.

The hardware and behaviour specification areas of the UML still require a significant amount of work to enable them to satisfactorily meet this requirement, as discussed below.

Behaviour Specification

A number of areas and components in the UML meta-model address the specification of behaviour:

- The Common Behaviour Package containing Actions and Signals etc.;
- The State Machine and Activity Packages containing state based behavioural concepts;
- The Collaboration Package containing interaction concepts; and
- The Use Case Package containing Actors and Use Cases etc.

Each of these groups of behavioural elements effectively addresses the specification of behaviour using a different set of concepts. They are all loosely linked to the structural specification elements of the meta-model (classes, relationships, etc.) but not consistent or integrated with each other, nor fully and unambiguously integrated with the structural elements.

For example, how is a particular Use Case related to the behaviour of other classes in the system, how does the activation of a Transition in a state machine relate to behavioural actions in the common behaviour package. There are associations defined that connect these components, but no clear explanation of the relationship between the semantics.

A single set of behavioural concepts should be identified, which can be mapped to different visualisations of that behaviour – either state based or interaction based. The visualisation should subsequently be exactly that: alternative visualisations of the same meta-model concepts, viewed differently to emphasise different aspects of the behaviour.

To achieve this it is necessary to identify the common behavioural concepts, between the different forms of behaviour visualisation. This in turn requires a clear understanding of two aspects of object behaviour. Firstly the generic behaviour (or semantics) of an object, in a possibly multi-threaded environment, and secondly how the concepts for defining specific object behaviour interact with the (structural) network of objects and with each other.

There is currently a body of work entitled “Action Semantics for UML” with an RFP ([OMG_98nov]) from the OMG. This may eventually address the behavioural problems with UML.

Hardware Specification

For a specific distributed system, the specification of the hardware used within the system is a significant part of the system design. The provision, within the UML, of Deployment diagrams and the concept of a Node does not adequately support the specification of the hardware technology forming part of a distributed system.

The definition of a Node as a subclass of Classifier should enable the specification of characteristics and a possible type hierarchy for hardware components. The notation descriptions in the literature do not show much use of these aspects – and it is thus not clear how such specifications should be constructed. The apparent lack of discussion of these aspects within the UML community raises the question as to whether the hardware specification concepts are adequate, unnecessary or simply so good that no aspect of their use raises significant interest.

The Permabase solution was to use class diagrams for hardware specification; however a preferable solution would be to use the UML as it is intended, with improvements to its hardware specification facility.

3.4.2.4 Need for an implementation model.

A system performance model predicts the performance of a particular system. A number of different systems could be implemented from a certain system design model. An implementation model defines a specific system, defining details about the number of instances of particular design components and their subsequent configuration and connectivity.

The details that distinguish one system implementation from another are information such as the number of clients connected to a particular server, the characteristics of those connections and the characteristics of the particular computing platform supporting the server. These detailed definitions are needed to characterise specific systems in order to predict their performance.

The UML facility for implementation model specification is significantly immature and does not satisfactorily meet this requirement.

The deployment and component diagrams are defined by the UML as implementation diagrams. However, they lack a clear description of how they should be used for the specification of (possibly large) distributed system implementation models.

One primary example is a clear definition of how the subsystem concept is applicable to implementation models. The concept of a subsystem exists within the meta-model as a subtype of both a classifier and a package and is defined to contain both “specification elements and realisation elements”. But how instances of such a defined subsystem are to be connected to other implementation components (e.g. Nodes) is not addressed at all.

Our solution to this problem is to introduce the concept of a subsystem containing a number of “access points”. An access point has a set of internal and a set of external connections. The internal connections are defined within the specification of the subsystem contents. When instantiated the subsystem is connected to its environment via the external connections of its access points.

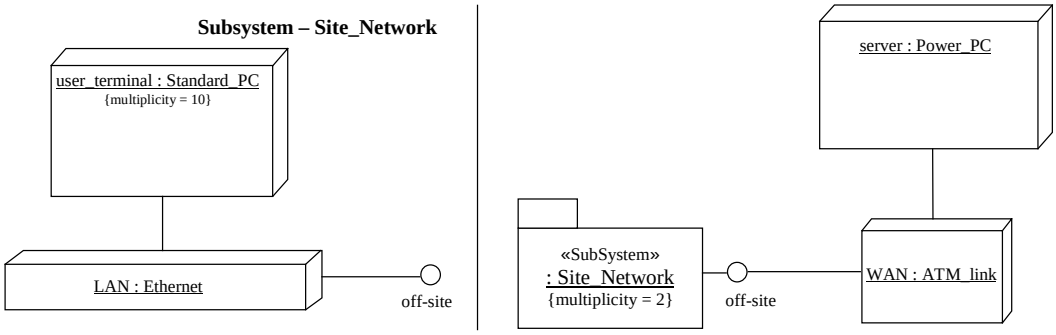

Figure 21 – Example Use of Access Point Connectors

Figure 21 shows an example deployment diagram using the notion of access points. The subsystem is defined to contain 10 user PCs connected via an Ethernet local area network (LAN). The LAN is connected to an access point that enables its connection to components external to the subsystem. The right hand side of the figure shows the instantiation of two instances of the subsystem, both of which are connected via the access point to an ATM wide area network (WAN). The WAN is connected to a powerful server PC.

As with the specification of hardware components, it is expected that the understanding of this area of the UML will improve and mature only if it gains increased use within the community. However, unless there is some attempt at a clear definition within the meta-model, this is unlikely to happen.

3.5 Summary

The Permabase project, which aimed to prove the concept that automatic generation of performance models from design specification, was overall successful. A prototype tool has been built that can be used to generate performance models.

However, although successful in concept, any future commercial tool to employ the techniques must acquire solutions to a number of problematic issues. These issues are listed in Table 1 along with an avenue for possible solutions.

Issue	Avenue for Solution
Translator specification and implementation techniques required.	Use the techniques proposed in this thesis.
Translations should be continuous/incremental. I.e. not require a long execution time.	The technique proposed in this thesis meets this requirement.
Results feedback not implemented, Translations need to be bi-directional, to aid feed back of results.	The technique proposed in this thesis meets this requirement.
UML (and its meta-model) needs to be more precise.	May be improved by UML version 2.0 ([OMG_99aug]) and in particular by the proposals of the precise UML group ([OMG_99dec]).
UML behavioural concepts require more precision and integration.	Improvements should follow from the Action semantics Request for Proposal ([OMG_98nov]) and submissions ([OMG_00aug]).

More functionally rich object-oriented database required to support the CMDS. Should include, configuration control, incremental changes + rollback, and enable efficient storage of variations on a base model.	No particular solution proposed, though many database engines exist that would meet at least some of these requirements.
Make better use of UML features for specifying Execution Environment, Workload, and System Scenario specifications. Enable more support for distributed system and multi-media features.	Solutions possibly by making use of UML profiles ([OMG_99jun], chapter 4). Ongoing work at University of Kent at Canterbury under the project “Design Support Environments for Distributed Systems”.

Table 1 – Permabase Issues and Avenues for Solutions.

Chapter 4

Translator Specification

The aim of this chapter is to provide a specification technique that is suitable for specifying translations between object-oriented models. It is a requirement that the specification technique uses the Unified Modelling Language (UML) language, in order that it is easily useable within a framework of other UML based specifications.

The approach to specifying model translators, introduced in this chapter, uses the UML and associated Object Constraint Language (OCL). UML Associations are used to define course-grained relationships between components from source and target models. The mappings defined by these Associations are refined with extra detail, where necessary, by adding OCL expressions in the context of the Association.

4.1 Introduction

Chapter 1 discussed some possible applications of a model translation process. In order to implement such applications it is necessary, as with any design process, to have a clear specification of what the translation process is intended to do. That is to say, a translator specification technique is required.

A translator is a function that takes one model, the source model, and creates a second model, the target model, based on the structure and information stored in the source model. For any particular source model, the target model should be unique.

Some applications require a translation in both directions; thus, each model takes on the role of source and target with respect to each of the translator functions. In this situation, the combination of the two translators can be seen as a translator relation, where each of the two models uniquely maps to the other. This relationship is particularly useful for applications such as that produced within the Permabase project, enabling information to be added to either model and reflected in the other; for example feedback of performance results into the system design specification.

An important issue is what to do if a translation cannot be performed. However, this is an issue related to the execution of actions and is hence left for discussion in relation to the implementation of translators. This chapter discusses the specification of translators in a declarative fashion – the specifications define what is or isn’t a valid translation, not how to perform the translation or what to do if it can not be performed.

In order to identify a general technique for specifying translators, it is first necessary to look at what the translators are translating between – in this case, object-oriented (OO) Models.

An OO model is a network of linked objects. Each object represents a component of the subject being modelled, and the links represent the relationships between the components.

This network of objects and links can be viewed as a graph, where objects are viewed as vertices and links viewed as edges. Using this view of an OO model, it is a logical step to attempt the application of Graph Transformation techniques for translating one OO model (graph) into another. Section 4.2 illustrates two different Graph Grammar based approaches to specifying model translation.

The traditional way of describing OO models is to use a set of Class specifications, often described using one of the many variations of 'Class Diagram'. Class Diagrams on their own are not such an expressive specification technique as Graph Grammars, however by adding a constraint language, a similar expressiveness can be achieved.

Section 4.3 of this chapter describes the technique for specifying model translators using the UML and OCL. Section 4.4 includes a general discussion regarding the style of specification employed by the UML/OCL technique, pointing out some interesting characteristics of this style. Section 4.5 discusses some related work and section 4.6 concludes the chapter.

4.1.1 The Example

The example used throughout this chapter is a translator for converting between a Directed Graph and a Tree data structure. In itself this could seem to be a very academic exercise, however, it can be seen as a generalisation and simplification of a number of more realistic translation problems, as discussed below.

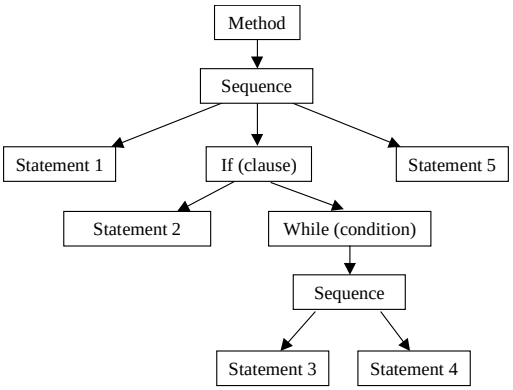

Figure 22 – An Abstract Behaviour Syntax Tree

```
Statement 1;
IF (clause) THEN {
  Statement 2;
} ELSE {
  WHILE (condtion) {
    Statement 3;
    Statement 4;

  }
}
Statement 5;
```

Table 2 – Pseudo Code

Within the Permabase project, part of one of the translation processes was to convert the specification of an object's behaviour into a Discrete Event Simulation (DES) model and into a Petri-net model of that behaviour.

The object behaviour is stored as an abstract behavioural syntax tree (e.g. as shown in Figure 22, or as pseudo code in Table 2) and the DES and Petri-net models are both extensions to the notion of a directed graph (e.g. as shown in Figure 23).

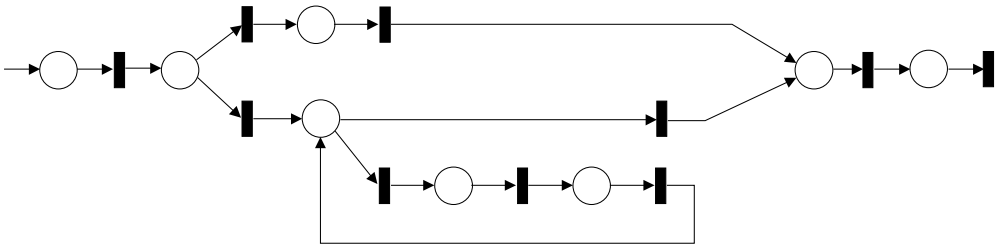

Figure 23 – A Petri-net (directed) graph

The Directed Graph ↔ Tree translator is a simplification of this task; in actuality, the translation doesn't have a one-to-one correspondence between nodes in the syntax tree and vertices in the Petri-net or DES models.

An alternative application of the Directed Graph ↔ Tree translator example is in the domain of visual languages. Editors for entering tree-like data structures are often based on a directed graph model to give the user more flexibility – allowing invalid trees to be specified as intermediate steps.

For example, an editor for specifying a class inheritance hierarchy would consist of class vertices and arrow edges that would allow loops in the input specification. The valid inheritance hierarchy would have to be a tree and a translator could form part of the editor implementation. This would involve the issue of what to do if an attempt is made to translate a directed graph into a tree when the graph does not map to a tree; i.e. if there were loops in the inheritance hierarchy. This issue is discussed further in the next chapter.

The Directed Graph ↔ Tree translator specification problem has a number of issues, but it is not the intention of this thesis to address or discuss the merits of one specification over another. This translator specification is simply used as an example to illustrate the specification technique.

4.2 Graph Grammar Approach to Model Translation

By viewing an OO model as a graph, we can look at Graph Grammar base specification techniques as possible means for specifying model translators. The following two subsections illustrate the Graph Transformation and Triple Graph Grammar approaches to specifying a translation.

4.2.1 Graph Transformation

The Graph Transformation approach for specifying a translation from one graph (model) to another is to conceptually combine the two graphs and define grammar rules that build the combined graph. The rules describing the source graph are taken as a starting point and are added to in such a way so that the target graph is created as though it was part of the source graph. The resulting target graph is subsequently a sub-graph of the combination.

To illustrate this, consider an example translation from a Tree model into a Directed Graph model.

A Directed Graph consists of Vertices and Edges and a Graph Grammar description of this is given in Figure 24, the interpretation of which is described below.

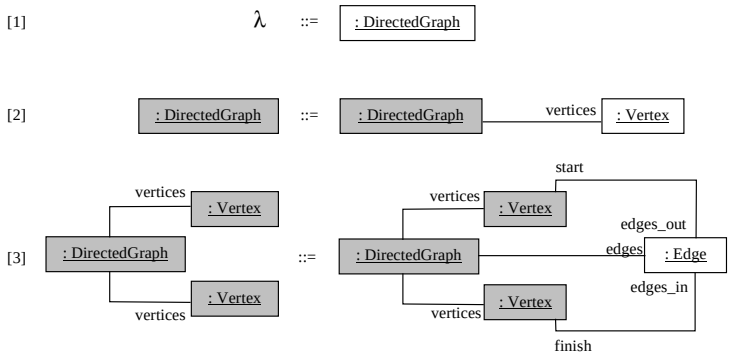

Figure 24 – Graph Grammar for Directed Graphs

Rule [1] states that a DirectedGraph object can be created as a starting point.

Rule [2] states that, given a DirectedGraph object, a Vertex object can be added.

Rule [3] states that, given two Vertex objects, an Edge object can be added, linking the two Vertex objects as shown.

A Graph Transformation grammar (or set of rewrite rules) is based on the grammar for the source graph, i.e. in this case on the grammar for Trees. The Directed Graph grammar is not explicitly used, though in this simple example traces of it are visible within the transformation rules.

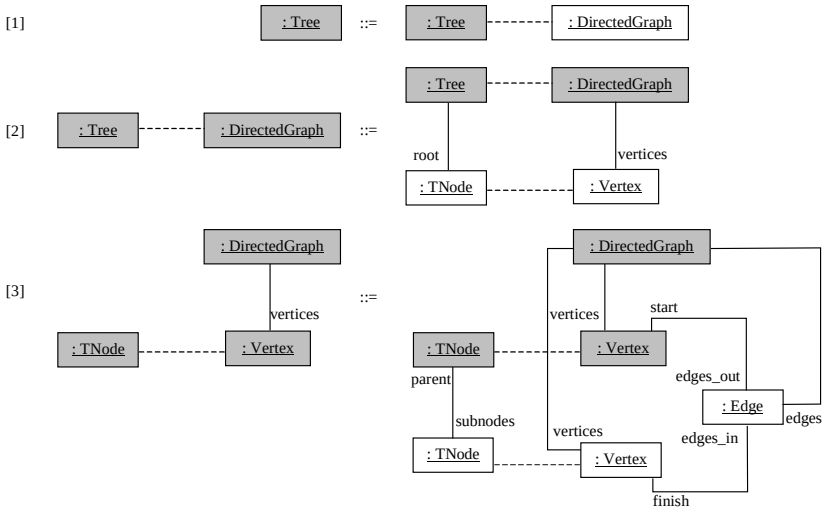

Figure 25 – Tree to Directed Graph transformation rules

Figure 25 shows the Graph Transformation rules for a Tree → Directed Graph translator. A description of the rules is given below.

Rule [1] states that given a Tree Object a DirectedGraph object can be created.

Rule [2] states that, adding a TNode object to a Tree means a Vertex object must be added to the Directed Graph.

Rule [3] states that, adding a TNode object as a subnode of another means another Vertex must be added to the Directed Graph, and it must be linked via an Edge object to the Vertex that represents the original TNode.

4.2.2 Triple Graph Grammars

Triple Graph Grammars (TGG) are an approach published in [Schürr_94jun] that extend the idea of Pair Graph Grammars [Pratt_71]. The TGG approach enables a clear distinction to be made between source and target graphs; it also keeps the extra links needed for specifying the transformations as a separate specification. Using this approach, separate grammars are produced for each of the graphs involved. The key is the production of a set of correspondence rules (the third grammar), which specify the homomorphic mapping between the two graphs.

A TGG specification is a declarative definition of the mapping between the two graphs and a translator in either direction can be implemented from the specification.

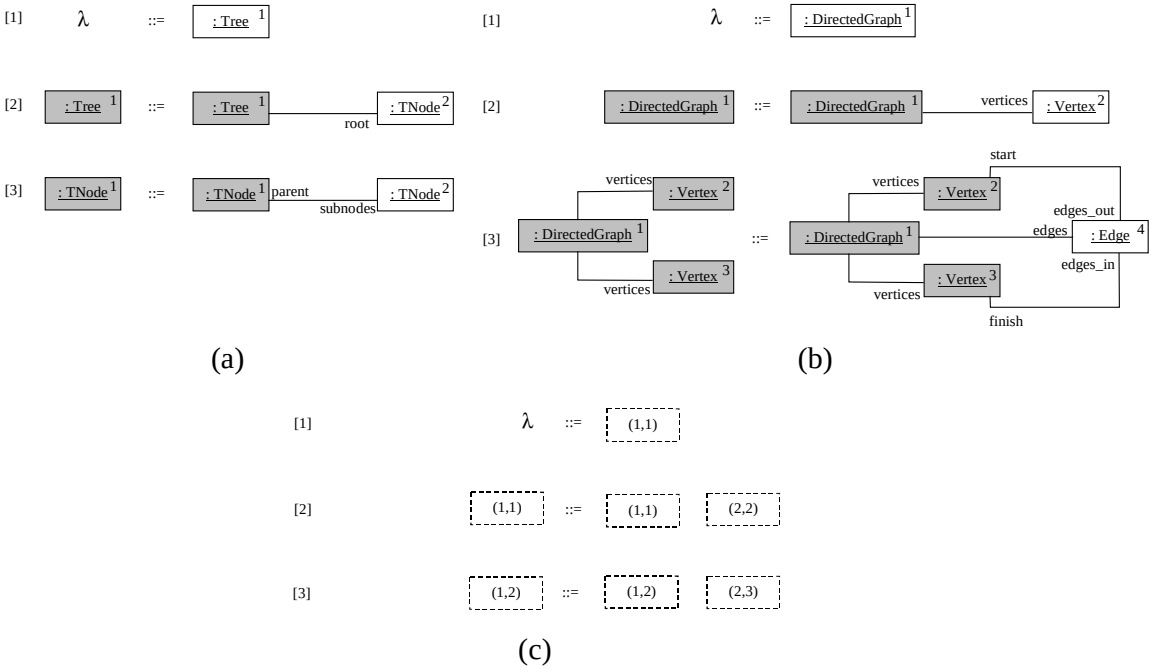

Figure 26 – Triple Graph Grammar based specification for Tree ↔ DirectedGraph Translation

Figure 26 shows a Triple Graph Grammar specification of the Tree↔Directed Graph translator. Figure 26a and Figure 26b show the grammars for the Tree and Directed Graph and Figure 26c shows the third grammar – the correspondence rules.

The vertices in the Tree and Directed Graph grammars are numbered and referenced by the correspondence rules. Each correspondence rule shows dashed vertices (x,y) that relate a vertex x in the corresponding left hand side (Tree) grammar rule and a vertex y in the corresponding right hand side (Directed Graph) grammar rule.

Effectively the correspondence rules define the extra links that are introduced to connect the two graphs when defining Graph Transformation rules for converting one graph into the other.

For example, rule 3 from the TGG shown in Figure 26 can be read from left to right showing how to generate an edge depicting the parent/subnode relationship between two Vertex/TNodes. Figure 27a shows the relationships between the three graph components, and Figure 27b illustrates the subsequent, combined, graph generation rule. See how this matches rule 3 of the Tree→DirectedGraph transformation rule defined in Figure 25. The rule should be read as follows:

"If there exists a Vertex/TNode pair, then adding a second TNode as a subnode of the first requires that a second Vertex be added along with an Edge directed from the first Vertex to the second."

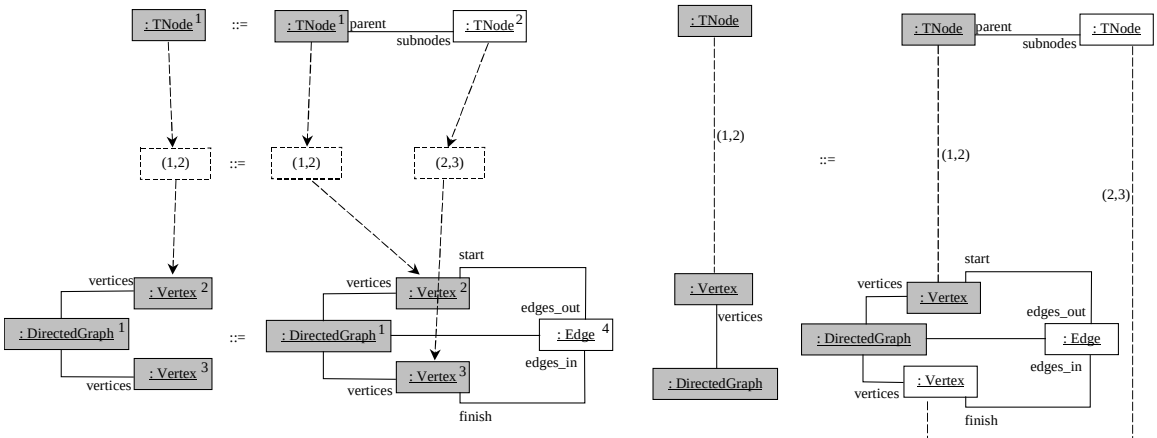

Figure 27 – Left → Right Interpretation of Figure 26 TGG, Rule 3

Or the rule can be read from right to left showing how a parent/subnode relationship in a tree should be created to reflect the addition of an edge to the directed graph. Figure 28a shows the relationship between the three TGG graphs and Figure 28b shows the subsequent, combined graph transformation rule. This rule should be read as:

"If there exists two Vertices, one of which is mapped to a TNode, adding an edge between these two vertices requires that a second TNode is created as a subnode of the first."

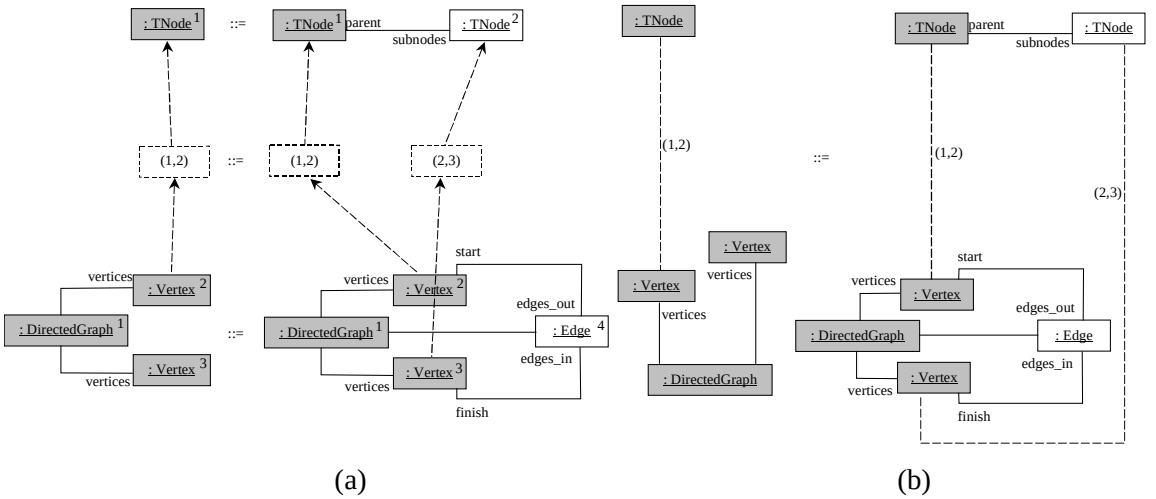

Figure 28 – Right → Left Interpretation of Figure 26 TGG, Rule 3

This technique enables the specification of each graph to be distinctly specified as separate grammars. The use of the third grammar defining the correspondence rules encapsulates the specification of the links that join the two graphs in a separate definition.

4.3 The UML/OCL Technique

The current standard language for OO design is the UML. This language (or family of languages) is widely understood within the OO community. Hence, it would be advantageous to be able to specify translators using this language.

The Graph Grammars define the set of graphs that validly match a certain specification. They achieve this for object models by defining valid patterns of object graph that may exist within the model as a whole.

Within the OO community there is an alternative technique for specifying valid patterns of object model – the class specification. Pictorially, class specifications are usually defined using a variant of the Class Diagram.

Class Diagrams are used across the OO community for specifying a required set of valid object models. The Class Diagram on its own has been recognised as not being expressive enough to capture all the constraints required for some model specifications, hence the OCL has been adopted as a means to extend the expressiveness ([Kleppe_etal_98]).

4.3.1 Example

As an example, the class specifications for the Tree and Directed Graph models are shown in Figure 30 and Figure 29.

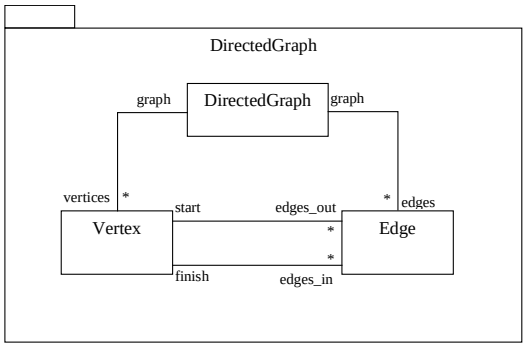

Figure 30 – Directed Graph Class Specification

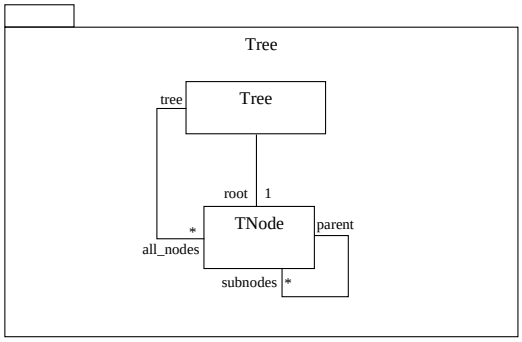

Figure 29 – Tree Class Specification

A *Directed Graph* consists of a number of *Vertices* and *Edges*; each *Edge* starts from one *Vertex* and finishes at another. An instance of the *Tree* data-structure contains one root *TNode*, which can contain a number of sub *TNodes* that in turn can contain sub *TNodes* etc.

In general the graph to tree mapping could be define such that:

- each graph Vertex maps to a tree TNode
- each Edge in the graph maps to a parent subnode link between two TNodes

By specifying the mappings informally, using English text, a reader can get an understanding of the mapping, but it is easy to leave out details or be ambiguous. For instance, the above descriptions do not describe that the direction of the *Edge* defines which *TNode* is the parent and which is the subnode.

A technique is required that allows formal checking of the specification and makes it easier to see if the constraints are incomplete.

4.3.2 A Translator Specification Architecture

The specification of each model can be grouped separately into two Packages. A 'Package' is a UML concept for enabling specifications to be defined in a modular fashion.

The specification of a translator between these two models can subsequently be defined by a third Package that 'uses' (depends on) each of the model definitions. This is illustrated in Figure 31.

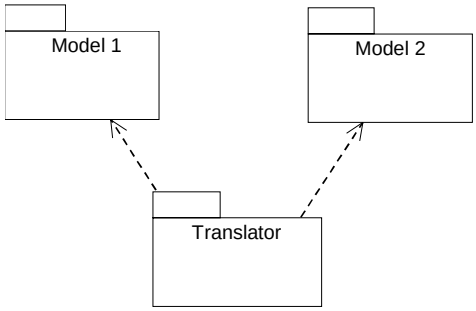

Figure 31 – General Architecture for Translator Specifications

The figure shows the specification of two 'Model' packages, both of which are 'used' by the Translator package. The dotted arrow indicates a dependency relationship between packages.

The example translator between a Tree and a Directed Graph is mapped onto this architecture as follows. The Model 1 and Model 2 packages must contain the definitions of the Tree and Directed Graph class specifications, and the contents of the Translator package must specify the mappings described in Table 3.

1. each graph *Vertex* maps to a tree *TNode* 2. each *Edge* in the graph maps to a parent subnode link between two *TNodes* 3. the direction of the *Edge* indicates the direction of the parent and subnode link

Table 3 – Mapping Specification

4.3.3 Translator Specification

The specification of the translator is a specification of the relationship or mapping between the types of component from each model, i.e. a specification of the mappings between classes from each of the two models.

Logically a translation is a process of converting one model into another. However, as the requirements of the Permabase project show, a translation in one direction often precedes the reverse translation back again. Hence, it would be advantageous if the translator specification was bi-directional.

That is to say that the translator specification should declaratively define the mapping relationship between the two models, rather than define how to create or generate one model from the other.

The specification of a mapping between one model and another can be broken down into the specification of mappings between the components of each model. The mappings between the components can subsequently be modelled as relationships between the components with a number of constraints imposed on them.

Adding a relationship between two components in UML is achieved by adding an association between the two related components. The constraints can then be written formally using a constraint language such as predicate logic, or as UML is being used, using OCL as a more compatible constraint language.

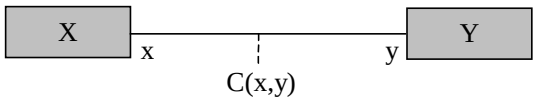

Figure 32 – A Mapping Specification (invalid UML)

Following this convention, a mapping specification would look like that shown in Figure 32. However, this is not strictly valid UML; the OCL constraint ("C(x,y)") is attached to the association, which is valid according to the UML meta-model, but the semantics of such a constraint are not defined.

Logically this is the correct place for the constraint, however as discussed in [Cook_etal_99] the UML 1.3 standard ([OMG_99jun]) is rather ill defined as to where OCL expressions may be specified. The standard only discusses the context of an OCL constraint in relation to an invariant on a *Classifier* or pre/post conditions on an *Operation*. Although, according to the meta-model definition, OCL expressions can form the body of a *Constraint*, which can be attached to any *ModelElement*.

Secondly, the semantics of a standard association are not quite the relationship between X and Y that is required. The specification is intended to define a relationship with the same semantics as a bijective binary relation as defined in set theory. However, there is currently no direct support for this type of relationship within the combination of UML and OCL.

4.3.4 Providing Support for Binary Relations

Specifically a binary relation R between two sets A and B is defined to be the set of pairs formed from the elements of A and B; expressed as follows:

$R : A \leftrightarrow B$ or
$R \subseteq AxB$

To define the particulars of this subset, the relation must be further constrained to select the required pairs from the Cartesian Product. For example, if we have a set of Parents (P) and a set of Children (C), then the relation F denoting fatherhood can be defined as follows:

$F = \{ (p,c) \mid p \text{ is male and } p \text{ is a father of } c \} \subseteq PxC$

Unfortunately, a Cartesian product (i.e. "AXB") is not directly expressible using OCL and neither is the concept of a relation or an Ordered Pair. It is this lack of Ordered Pairs that is the fundamental problem. This and other issues related to the expressive power of OCL are discussed in some depth by the authors of [Mandel_Cengarle_99].

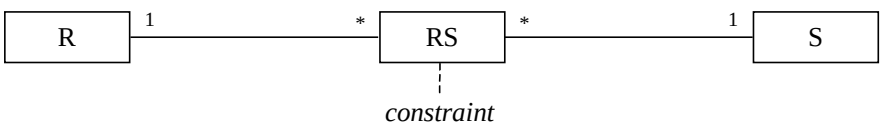

Figure 33 – Supporting Class for a Cartesian Product ([Mandel_Cengarle_99])

They suggest that the solution to defining a Cartesian product is to define a class that represents the required type (e.g. as shown in Figure 33). The following constraint[6] must subsequently be attached to the defined class to ensure that the set of instances of the class does in fact form the Cartesian product.

```
R.allInstances->union(S.allInstances)->forAll( r,s : oclAny |
  if r.oclType.name == s.oclType.name then
    true
  else
    RS.allInstances->exists( t : RS |
      t.r == r and t.s == s)
  endif
)
```

The constraint defines that there must be an instance of class RS for every pair of objects taken from the union of the instances of R and S, when the paired objects are of different types.

However, there are problems with this definition. If there is a subtype relationship between classes R and S then some of the instances of R and S are lost when the union is formed; hence, some of the members of the Cartesian Product will not be included.

Mathematically a Cartesian Product is formed as follows:

By defining the notion of a pair formed from elements of two sets; i.e. if there are sets A and B and x is an element of A and y is an element of B then the pair consisting of x and y is denoted as:

(x,y)

For any two such pairs, if the co-ordinates are the same then the pairs are equal, i.e. as shown below:

$(a,b) = (x,y) \Leftrightarrow (a=x \text{ and } b=y)$

This notion of pair is subsequently used to define a Cartesian Product AxB as the set of all pairs where the first co-ordinate is taken from A and the second taken from B, i.e. as shown below:

$AxB = \{ (x,y) \mid x \in A \wedge y \in B \}$

To do the same thing using the concepts of UML and OCL, a class *Pair* is defined with a constraint defining the equality of two pairs to depend on the equality of the members of the pair. This is shown in Figure 34.

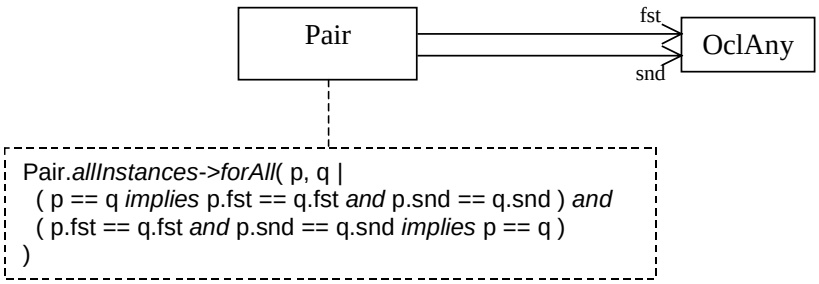

Figure 34 – Definition of Pair

The *Pair* class is extended with the definition of a parameterised *CartesianProduct* class as shown in Figure 35.

[6] Taken from ([Mandel_Cengarle])

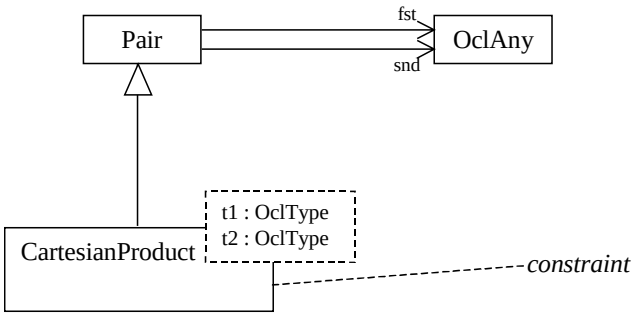

Figure 35 – Support for Cartesian Products

The *CartesianProduct* class must have an attached constraint that ensures that all instances of the class exist in accordance with the instances of the classes passed as parameters. The parameterised class defines the Cartesian Product t1 x t2 and the invariant constraint on the class must therefore be defined as follows:

```
t1.allInstances->forAll( x |
  t2.allInstances->forAll( y |
    CartesianProduct<t1,t2>.allInstances->exists( p |
      p.fst = x and p.snd = y
    )
  )
) and
CartesianProduct<t1,t2>.allInstances->forAll( p |
  t1.allInstances->exists( x |
    t2.allInstances->exists( y |
      p.fst = x and p.snd = y
    )
  )
)
```

The first part of the constraint ensures that all members of the Cartesian Product exist as instances of the class *CartesianProduct*. The second part ensures that new instances of the class *CartesianProduct* can not be created from instances of A and B in such a way that they form an object that is not a member of the Cartesian Product of the instances of A and B.

Any particular Cartesian Product can be formed by using a ‘Bound Element’ – a class definition that provides values for the parameters of a ‘Parameterised Class’. For example, Figure 36 shows a Cartesian Product of classes A and B.

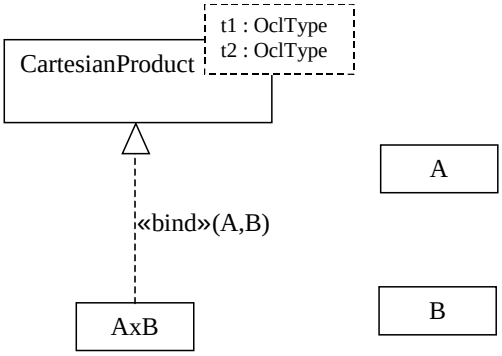

Figure 36 – Formation of a Cartesian Product AxB

The constraint on *CartesianProduct* ensures that only appropriate instances of AxB exist.

4.3.5 Constructing a Bijective Relation

The required semantics of our mapping relationship is that each instance of the class at one side of the association must have a one-to-one correspondence with an instance of the class on the other side of the relationship. This is known as a bijective relation.

A bijective relation can be supported in UML by extending the *CartesianProduct* class and adding an extra constraint defining the required properties.

To define these properties on a relation, firstly we define that the relation is a function (F), as show below:

$$F = \{ f \subseteq AxB \mid \forall p \in f\ (\exists x \in A\ (p.fst = x) \wedge (\forall p,q \in f\ (p.fst == q.fst \Rightarrow p == q) \}$$

Secondly, this constraint is extended, defining that a bijective relation is a function for which the inverse relation is also a function. This can be specified as shown below:

$$BjM == \{ f \subseteq AxB \mid \forall p \in f\ (\exists x \in A\ (p.fst = x) \wedge (\forall p,q \in f\ (p.fst == q.fst \Rightarrow p == q) \wedge \forall q \in f\ (\exists y \in A\ (q.snd = y) \wedge (\forall p,q \in f\ (p.snd == q.snd \Rightarrow p == q) \}$$

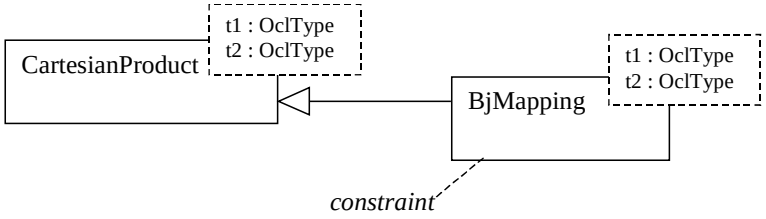

Figure 37 – UML Specification of a Mapping Relationship

Figure 37 shows the definition of a *BjMapping* class using UML; it extends the *CartesianProduct* and consequently is also a parameterised class, requiring the two class type parameters to be bound. The constraint on the *BjMapping* class is defined as follows:

```
BjMapping.allInstances->forAll( p, q |
  t1.allInstances->exists( x | p.fst == x) and
  t2.allInstances->exists( y | q.snd == y) and
  (p.fst == q.fst implies p == q) and
  (p.snd == q.snd implies p == q)
)
```

Using this parameterised class, specific mappings can be defined between arbitrary pairs of classes. Figure 38 shows an example, defining a mapping between classes A and B.

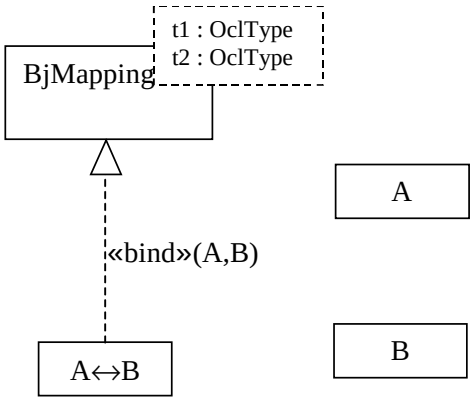

Figure 38 – Specification of a BjMapping between classes A and B

Mathematically, a bijective relation cannot always be defined between two sets. Consequently, a bijective relation can only be formed from classes A and B if the sets of instances of those classes are such that the constraints on the *BjMapping*, *CartesianProduct* and *Pair* classes are not invalidated. Subsequently, any new instance of classes A or B must be created along with the corresponding A or B instance so that the constraints are not invalidated.

4.3.6 Semantics of the mapping relationship

Given these supporting definitions a stereotyped association can be defined with the appropriate semantics. Going back to the original mapping specification shown in Figure 32, all that is necessary to convey the correct semantics is to add a stereotype label to the association, as illustrated in Figure 39.

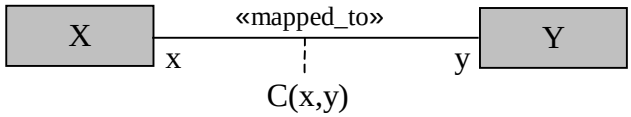

Figure 39 – A Mapping Specification (valid UML)

Given this specification the semantics of the stereotype are defined to imply the specification shown in Figure 40.

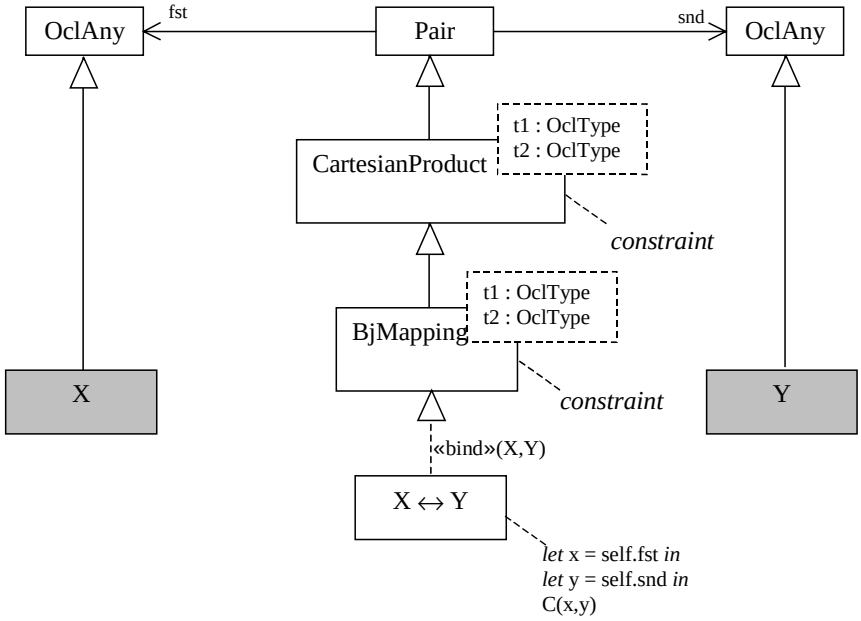

Figure 40 – Full expansion of the specification shown Figure 39

This expansion of the specification defines firstly, a binding of the BjMapping class that creates a specific mapping class between the two classes from each of the ends of the association – the class $X \leftrightarrow Y$.

Secondly, the constraint attached to the association is interpreted as a constraint on the specific mapping class. To ensure that references within this constraint validly refer to the mapped objects two ‘let’ expressions (enclosing the constraint) assign the rolenames of the association to be the two objects connected. These are respectively the objects fst and snd of the mapping inherited from *Pair*.

4.3.7 One-To-Many or Many-To-Many Mappings

In many cases it is not sufficient to specify only one-to-one mappings between the components from each model, more often 1-to-n or n-to-m mappings are required. (A 1-to-n mapping is a special case of n-to-m and does not need to be separately discussed.)

To specify these graphically an n-ary association[7] is used. This does however, cause a problem with the semantics and subsequent interpretation of the specification. The 1-to-1 case makes use of the fact that the association has two ends – one for a component from each model. With an n-ary association however, there are as many ends as there are components involved in the association.

A solution to this problem would be to make the distinction based on the names of the components involved. Each component is drawn from one or other of the two models over which the mapping is being specified – thus it is simple to deduce which components involved in the n-ary association are from which model. However, this requires a reader of the specification to refer to the model definitions in order to interpret the specification and gives no means for a precise semantic interpretation.

Another solution, giving visual indication more locally to the specification would be to "tie" the association ends together for components from the same model. This technique makes use of valid UML syntax, though the exact semantic meaning of it is not clearly specified within the UML standard ([OMG_99jun]). Section 3.41 of the standard shows use of the "tie" for defining an *XOR* association; it is used here to define a grouping of association ends.

Figure 41 illustrates the graphical specification of a general n-to-m mapping between components *V,W* and *X* from one model and components *Y* and *Z* from another, with the constraint *C(v,w,x,y,z)* on the mapping.

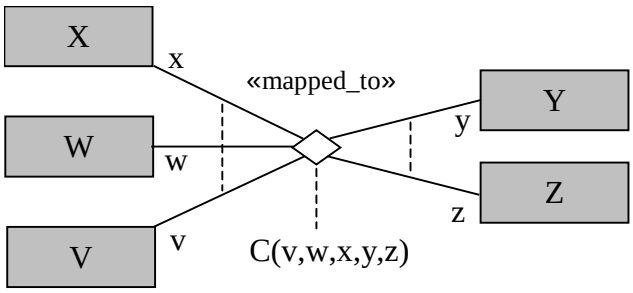

Figure 41 – General Mapping Specification

In Figure 41 the dotted lines between the two pairs of association ends indicate that classes *V,W* and *X* come from one model, and classes *Y* and *Z* come from the other. Thus, the mapping class relates two tuples of objects – a 3-tuple from VxWxX to a pair from YxZ.

The semantic interpretation of this is shown in Figure 42. A BjMapping class is defined between the *nTuple* and *Pair* classes, with the attached constraint equal to the constraint attached to the n-ary association, preceded by a number of let expressions and type checking constraints.

[7] See Chapter 2 for a description of UML concepts such as n-ary association.

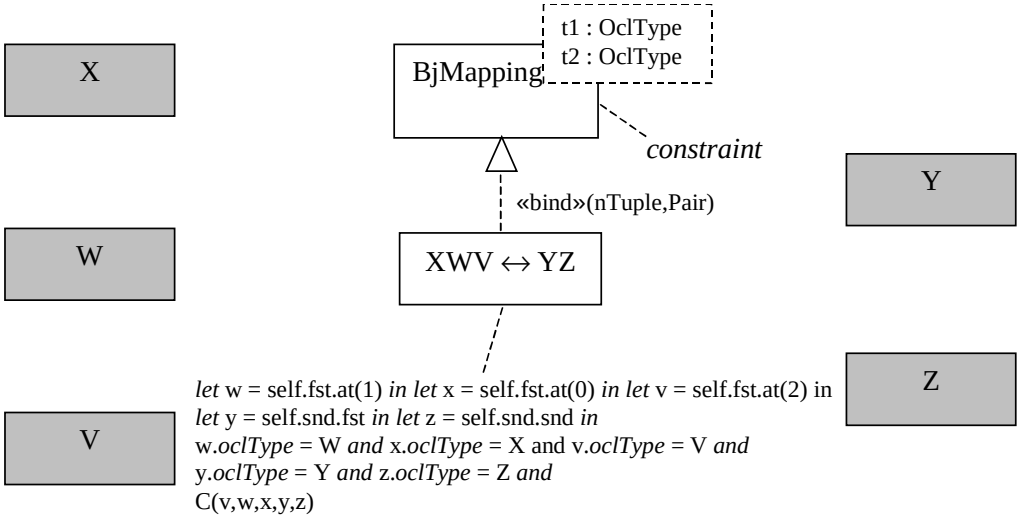

Figure 42 – Semantic interpretation of Figure 41

The let expressions, as with the one-to-one case, define the role names of the n-ary association to refer to the mapped objects. The type checking constraints are required due to the use of the un-typed *nTuple* and *Pair* classes for grouping the objects on either side of the mapping.

4.3.8 «mapped_to» operator

Within the specification of the constraints defined on a particular mapping, it is sometimes necessary to refer to a mapping between other components. To enable this an operator is introduced to OCL that facilitates succinct reference to mappings.

The operator is given two forms of syntax, '↔' or '«mapped_to»' for use if the more graphical symbol is unavailable. The operator can be used as follows, in three different cases:

4.3.8.1 Object «mapped_to» Object

This is the one-to-one case of one object mapped to another; it specifies that a mapping must exist between the two objects. More formally, the semantics of the operator can be interpreted as follows:

o1 ↔ o2 ≡ BjMapping<o1.*oclType*,o2.*oclType*>.*allInstances*->*exists*(m |
m.fst == o1 *and* m.snd == o2)

4.3.8.2 N-tuple «mapped_to» M-tuple

This is the n-to-m case of mapping *n* components to *m* other components, its interpretation is similar to the one-to-one case. Ignoring, for a moment, that tuples are not expressible in OCL, a 'n-tuple ↔ m-tuple' specification defines that a mapping object must exist that relates these two tuples.

To get around the problem of representing tuples in OCL either, a Sequence can be used, or a specifically sized tuple class can be defined (as discussed in [Akehurst_Bordbar_01]). The second method is preferred as this avoids confusion with the third use of the operator (see below).

The definition of an explicit Sequence is by using curly braces preceded by the term 'Sequence' as shown in the example below (see section 7.5.12 of the UML standard):

Sequence { a, b, c }

Thus, if there are tuple classes defined, and OCL can be extended to allow definitions of explicit user defined classes (similarly to the explicit definition of Sets, Bags and Sequences) an expression:

(a, b, c) ↔ (x, y)

can be interpreted as:

```
BjMapping<n3Tuple, Pair>.allInstances->exists( m |
  m.fst = n3Tuple { a, b, c } and
  m.snd = Pair { x, y }  )
```

4.3.8.3 Sequence «mapped_to» Sequence

This third use of the operator is for referring to several mappings. It should be interpreted to mean that there is a bijective relation mapping each object in one sequence to an object from the other sequence. This can be expressed formally as shown below:

```
seq1 ↔ seq2 ≡ seq1->size == seq2->size and
                Sequence {0..seq1->size} ->forAll( i |
                  Let a = seq1->at(i) in
                  Let b = seq2->at(i) in
                  BjMapping<a.oclType,b.oclType>.allInstances->exists(m |
                    m.fst == a and m.snd == b ) )
```

If the operator is used between collections that are not specifically Sequences, then the predefined OCL ‘asSequence’ operation can be used.

4.3.9 Well-formed Mappings

When creating a mapping specification, the issue of its ‘well-formedness’ must be addressed; i.e. is the mapping a valid specification.

A mapping relates a number of classes and constrains the relationship using the attributes of the related classes. Hence, if an attribute or rolename is used in the constraints specified for a mapping, then the class from which that attribute or rolename is navigable must be related as part of the mapping. (This becomes particularly relevant with respect to implementing the mapping, see next chapter.)

To illustrate this, Figure 43 shows a mapping between the DirectedGraph and Tree classes. It would seem to be an obviously correct relationship between the two classes. However, by analysing the constraints it can be seen to be ill-formed.

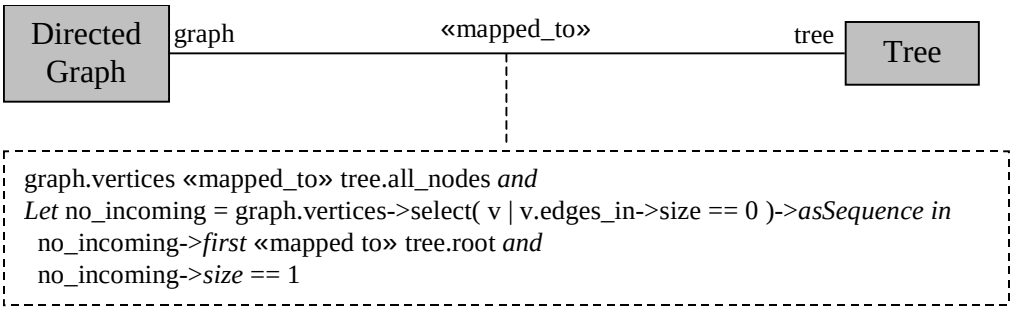

Figure 43 – Ill-formed Mapping Specification

As the containing objects for each of the mapped models, it is correct to create a mapping between these two classes. The constraints refine the mapping to ensure that:

1. Each Vertex is mapped to a TNode, and
2. One of the Vertices with no incoming Edge is mapped to the root of the Tree, and
3. That there is only one Vertex with no incoming Edge.

The first part of this constraint, uses rolenames ‘vertices’ and ‘all_nodes’ which are navigable from the DirectedGraph and Tree classes (see Figure 45 below).

The second two parts of the constraint use ‘no_incoming’, defined by the *Let* expression to be the Sequence of vertices with no incoming edges. The definition of no_incoming uses the rolename ‘edges_in’, which is navigable from a Vertex not from the DirectedGraph class; thus its use in the constraint implies that the Vertex class must form part of the mapping.

Through thinking about the purpose of the second two parts of the constraint, it can be seen that this does make sense. They define a mapping between one of the Vertices in the graph and the root of the Tree. In order to determine which Vertex to use, it follows that all of the Vertices must be looked at to see if they are valid candidates for mapping to the root.

There is no way to form the constraint that only refers to attributes or rolenames of the DirectedGraph and Tree classes. One could use an alternative form that used rolenames of the Edge class instead of the Vertex class, e.g.:

Let no_incoming = (graph.vertices – graph.edges.finish)->*asSequence in*

Here, the rolename ‘finish’, navigable from the Edge, class is used and hence all of the Edges in the graph must form part of the mapping instead of all of the Vertices.

Using the original form of the *Let* expression, the well-formed mapping is as shown in Figure 44.

A collection of instances of the Vertex class is included in the mapping; this collection is defined by an additional part of the constraint to be the set of vertices contained in the graph.

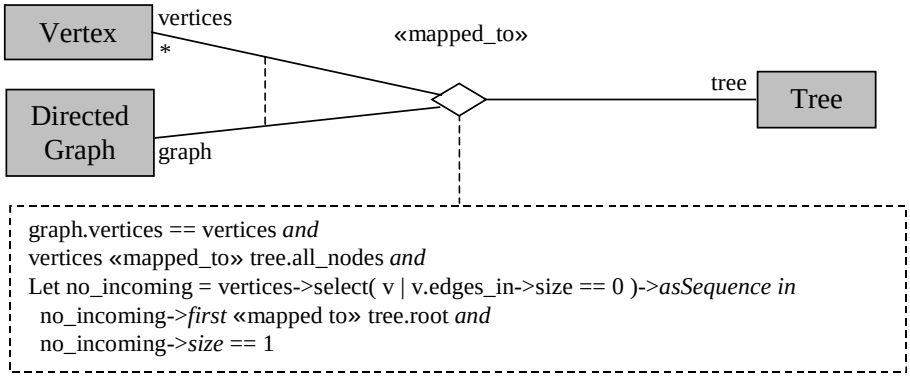

Figure 44 – Well-formed Mapping Specification

4.3.10 Example

Using the specification ‘language’, as explained in the previous subsections, the definition of our example Translator can be written as shown in Figure 45.

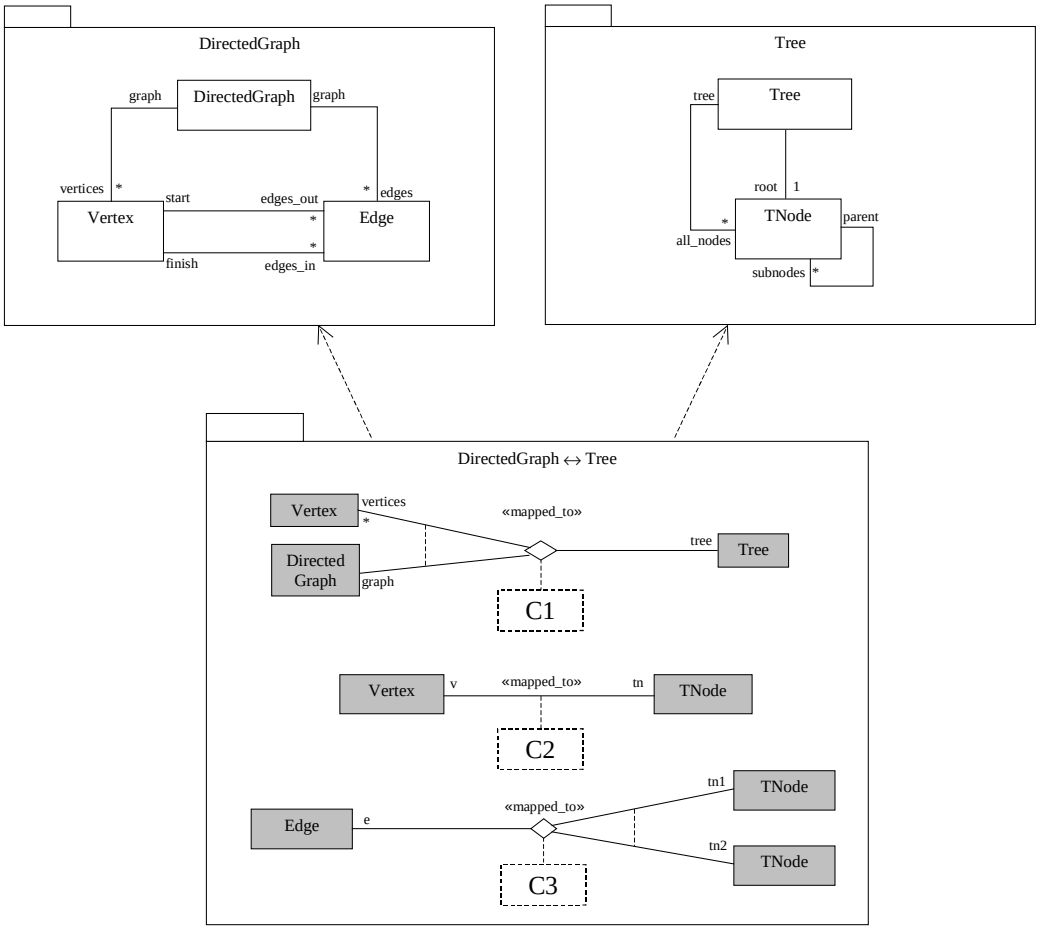

Figure 45 – UML/OCL specification of a Tree↔Directed Graph Translator

The figure shows the three packages containing the different parts of the translator specification architecture. The top two packages are the distinct definitions of the two models and the bottom package contains the mapping specifications that define the translator.

The constraints C1 to C3 that are specified in the DirectedGraph↔Tree Translator are as shown in Table 4, along with an OCL comment explaining the constraint.

C1	-- All vertices are mapped to TNodes and the graph must contain only one vertex with no incoming edges, which is mapped to the root of the tree. **graph.vertices == vertices *and*** **vertices «mapped_to» tree.all_nodes *and*** ***Let* no_incoming = vertices->*select*(v | v.edges_in->*size* == 0)->*asSequence in*** **no_incoming->*first* «mapped to» tree.root *and*** **no_incoming->*size* == 1**
C2	-- All edges starting from the Vertex are mapped to TNode pairs, where the fst of the pair is the TNode mapped to the Vertex, and the second is a subnode of the TNode. **v.edges_out «mapped_to» tn.subnodes->*iterate*(ts; *acc*:*Set* |** ***acc*->*includes*(Pair{tn, ts})**
C3	-- Each edge is mapped to the relationship between two TNodes, the TNode mapped to the Vertex at the end of the Edge is a subnode of the TNode mapped to the Vertex at the start of the Edge. **e.start «mapped_to» tn1 *and*** **e.finish «mapped_to» tn2 *and*** **tn1.subnodes->*includes*(tn2)**

Table 4 – Constraints for DirectedGraph ↔ Tree Translator

Using this specification it is possible to determine whether or not two model instances (i.e. a DirectedGraph and a Tree) are valid translations of each other. However, as stated in the introduction, the specification does not describe what should happen if the models are not valid translations.

For example, the graph shown in Figure 46 does not translate into a valid tree (Figure 46a shows a visual representation of the graph, Figure 46b shows an object diagram representation of the graph).

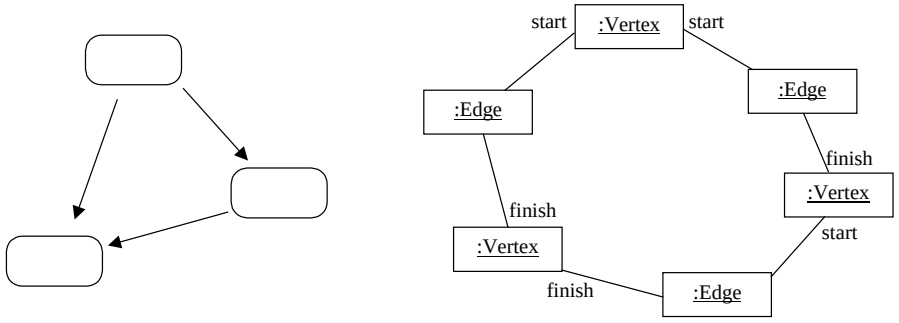

Figure 46 – An Example DirectedGraph

Each of the trees shown in Figure 47 could be a possible interpretation of the graph – if one or other of the graph edges were removed. The positions of the respective translated removed edges are shown dashed.

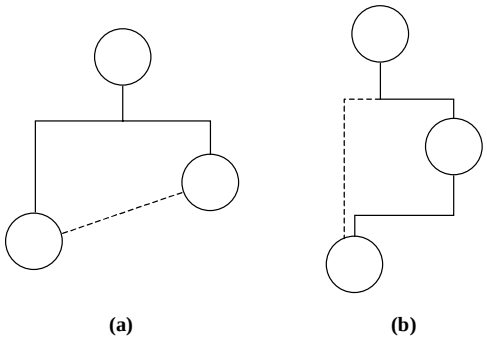

Figure 47 – Possible Tree Translations of Figure 46

It is not the purpose of the translator specification to define which (if either) of these trees is the correct translation, the specification simply states that there is not a valid translation between the graph and either tree.

4.4 UML/OCL Specification Style

The proposed UML/OCL translator specification technique is a *Graphical*, *Object-oriented*, *Bi-directional*, *Declarative* specification of the relationship between two object models. This section discusses each of these characteristics, enabling the reader to determine whether the specification technique is appropriate for a task they have in mind.

4.4.1 Graphical

The mapping specifications that form a translator specification are defined using a standardised Graphical language – the UML. In contrast, another standardised

language – XSLT [W3C_99nov] – can be used to specify (or even implement) a translation from one XML model to another; however, XSLT is a textual language.

The graphical nature of UML gives a high-level view of the components from each of the related models and their relationship, which enables immediate recognition of the association between components. The details of which can be subsequently derived from the attached OCL constraints.

An XSLT specification, in contrast, is directly executable. However, its textual nature means that it is more complex to interpret at a high-level by a human reader. An example XSLT specification defining the translation from an XML model of a Tree into an XML model of a Directed Graph is included in Table 5. This can be compared with the UML/OCL specification found in Chapter 4 and Appendix D.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/TREE">
  <GRAPH>
    <xsl:text>&#13;&#10;  </xsl:text>
      <VERTICES><xsl:text>&#13;&#10;</xsl:text>
        <xsl:apply-templates select="//TNODE" />
        <xsl:text>  </xsl:text>
      </VERTICES>
    <xsl:text>&#13;&#10;  </xsl:text>
      <EDGES><xsl:text>&#13;&#10;</xsl:text>
        <xsl:apply-templates select="//SUBNODES" />
        <xsl:text>  </xsl:text>
      </EDGES>
    <xsl:text>&#13;&#10;</xsl:text>
  </GRAPH>
</xsl:template>

<xsl:template match="TNODE">
  <xsl:text>    </xsl:text>
  <VERTEX>
     <xsl:attribute name="id">
       <xsl:value-of select="generate-id()"/>
     </xsl:attribute>
     <xsl:text>&#13;&#10;</xsl:text>
     <xsl:text>      </xsl:text>
     <DATA><xsl:value-of select="DATA"/></DATA>
     <xsl:text>&#13;&#10;    </xsl:text>
  </VERTEX>
  <xsl:text>&#13;&#10;</xsl:text>
</xsl:template>

<xsl:template match="SUBNODES">
  <xsl:for-each select="TNODE">
    <xsl:text>    </xsl:text>
      <EDGE>
        <xsl:attribute name="from">
          <xsl:value-of select="generate-id(../..)"/>
        </xsl:attribute>
        <xsl:attribute name="to">
          <xsl:value-of select="generate-id()"/>
        </xsl:attribute>
      </EDGE>
    <xsl:text>&#13;&#10;</xsl:text>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Table 5 – XSLT for Tree to Graph Translation

4.4.2 Object Oriented

The use of UML as a specification language enables the use of the technique alongside the specification of models using the current industry standard and Object-Oriented approach to software engineering. Irrespective of the merits or problems

regarding Object-Oriented specification and the use of UML, they are in current widespread use within the industrial community.

The use of UML as a language for specifying model translators enables the use of translators as a concept to be more seamlessly integrated with the specifications of other aspects of a software system. Hence, this will enable translators to become more widely adopted as a software engineering technique.

Alternative specification techniques, such as Graph Transformations, are not Object-Oriented. Although having a long history and large supporting research community, Graph Transformations have not been as widely adopted by industry (although there has been and still is some use). Figure 48 shows a Graph Transformation based specification for the Tree to Graph translator example.

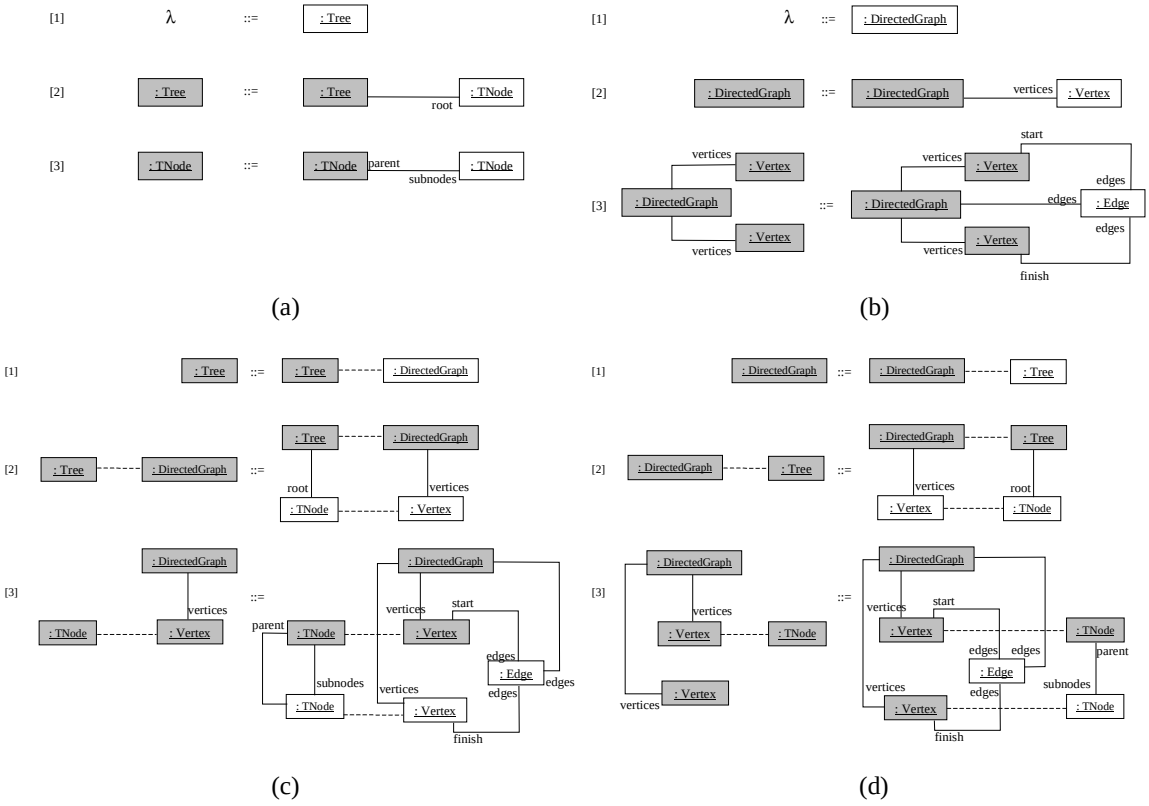

Figure 48 – Graph Transformation based specification for Tree ↔ DirectedGraph Translation

Parts (a) and (b) of the figure show a Graph Grammar for the Tree and DirectedGraph models. Part (c) defines a set of Graph Transformation Rules for building a DirectedGraph from a Tree, and part (d) defines the inverse rules for building a Tree from a DirectedGraph.

Although there is a long history of research within the Graph Grammar community it is only recently that knowledge and use of them has become more wide spread. As stated in [Bardohl_etal_99] (in the context of Graph Transformations used within the domain of Visual Languages) Graph Transformation techniques suffer from a "scaling-up problem" and a lack of efficient tools.

The gradual improvement and development of Graph Grammar based tools (such as that described in [Schürr_etal_95]) has been a contribution towards the expansion of the Graph Grammar Community.

4.4.3 Bi-Directional

The UML/OCL specification technique is bi-directional. In reading or writing a translator specification using this technique, no assumption is made as to which is the source model and which is the target model. Either side of the specification can be source, target or both depending on how the specification is applied.

The UML/OCL translator specifications are a specification of the required relationship between two models, rather than a specification of how to generated one model from the other. In contrast, both the XSLT and Graph Transformation approaches are unidirectional. The specifications make a clear distinction between source and target models and explicitly define a set of rules describing how one model should be created from the components of the other.

The Triple Graph Grammar (TGG) approach developed by Andy Schürr [Schürr_94jun] is an adaptation of the Graph Transformation technique that provides a bi-directional specification. The technique as been discussed in Chapter 4 and can be seen to provide a comparable, bi-directional and graphical specification of a translator (or transformation). The main drawback of the approach, with respect to the requirements addressed by this thesis, is that it is not Object-Oriented.

Both the UML/OCL and TGG bi-directional techniques provide a concise specification of a translation relationship between two models. Single direction translators can be implemented from the bi-directional specifications and additionally (as described in Chapters 5 and 6) bi-directional translators can be implemented.

4.4.4 Declarative

A UML/OCL translator specification is declarative. It does not specify how to create one model from another, the specification defines what conditions must be met by each model in order that they are valid translations of each other.

The classic distinction between declarative and imperative programming languages bases the distinction on the evaluation of expressions in the language. Declarative expressions evaluate to true or false, where as imperative expressions may be commands or questions and may evaluate to any value or none. An imperative program is a sequence of commands that defines how to solve a given problem. A declarative program is a set of expressions that define what the problem is; how it is solved is left to the interpretation of the expressions.

According to this definition, each of the techniques mentioned in this thesis, are declarative. The XSLT and Graph Transformation techniques are evaluated by using a complex pattern-matching algorithm that detects instances of a defined (Left Hand Side, LHS) pattern and replaces it with an instance of an alternative (Right Hand Side, RHS) pattern.

In contrast, the UML/OCL technique does not define replacement rules. Although there are notionally two sides (one for each model), they are not defining classical grammar-like ‘Production Pairs’. The mapping definitions don’t define patterns to be replaced by other patterns.

Primarily, the difference is the semantics of the components used in the specification. UML/OCL translator specifications are constructed from Classes and Associations; however, the models that they are defining a translation for, are constructed from Objects and Links. With both the XSLT and GT specifications, the rules are defined in terms of the elements that make up the models, i.e. nodes and edges or XML elements.

The UML/OCL specification uses a higher-level abstraction of the model elements in order to define the translation. Classes and Associations are abstractions of Objects and Links that define valid patterns of Objects and Links for a particular model.

Pattern recognition is still forms part of the overall interpretation. Rather than being part of interpreting the translator specification (transformation rules), it is an integral part of the semantics of the model definition language (Class Diagrams). The interpretation of the translator specification does not require a separate pattern-matching element; it can be understood at the same abstract-level (meta-level) as a Class Diagram specification.

Consequently, various sets of refinement rules can be defined that map the abstract (UML/OCL) translator specifications to a number of alternative lower-level definitions, such as:

- an XSLT specification;
- a set of Graph Transformation rules;
- a one way translator implementation (see Chapter 5); or
- an active (two-way) translator implementation (see Chapter 5 and 6).

4.5 Related Work

The authors of [Fischer_etal_99] describe a new graph grammar language called “Story Diagrams”. The authors identify the problems with graph grammar notations and the lack of integration with the OO philosophy and propose Story Diagrams as a solution to these problems.

Story Diagrams use a combination of UML class diagrams, UML activity diagrams and UML collaboration diagrams to represent graph grammars and graph rewriting rules. The semantics of Story diagrams are based on Progres; earlier work documented in [Schürr_etal_95], [Schürr_97] and others.

This work is complimentary to the technique proposed in this chapter, adding confidence to the proposed use of UML as translator specification tool.

Other work, by Jahnke and Zündorf ([Jahnke_etal_96], [Jahnke_Zündorf_98] and [Jahnke_Zündorf_99]), also builds on Progres. Their work makes use of the Triple Graph Grammar approach to enable the implementation of a design environment for specifying translators between relational and object-oriented database schemas.

4.6 Conclusion

This chapter has illustrated three different techniques for specifying translations between object-oriented models. Two of the techniques are based on Graph Grammar theory and non-standardised notations. The main part of this chapter has described a UML and OCL based technique for specifying model translators. All of the techniques have been used to define the same example translator specification.

The UML/OCL technique is not intended as a replacement for the Graph Grammar based approaches. Nor is it claimed to be better in any fashion, although there may be advantages and disadvantages of each technique.

The proposed technique is a natural way of defining a two-way mapping between two distinct models within an object-oriented modelling domain. The use of UML and OCL to describe the mappings integrates seamlessly with the Object Management Group's Model Driven Architecture initiative, enabling UML to be used as a language for specifying model translations.

4.6.1 Future Work

A useful area for future research would be an investigation into how to conclude if the mapping specifications are complete. I.e. to answer the question "Given a set of defined mappings can any instance of one model be mapped into the other?"

With respect to the Permabase project (see chapter 3), this technique could be used to define the translation from the concepts of the abstract design model into a model of other concepts suitable for a particular analysis engine. This technique defines a two-way mapping, hence aiding the implementation of feeding back the results of analysis into the abstract system model and subsequently to the concrete representation.

A possible extension to the specification technique would be to enable the use of the association between classes to reduce the complexity of some mapping relationships. For instance, the DirectedGraph↔Tree translator specification defines a mapping between an Edge class from the graph model and two TNodes from the tree model. It may be possible to specify the mapping as a relationship between the Edge class and the parent/subnode association between the two TNodes.

Chapter 5

Translator Implementation

The previous chapter has discussed techniques for specifying model translators and indicated a number of advantages for using the UML/OCL based technique. If this approach to specifying translators is adopted, an implementation technique that is compatible with it is required.

This chapter illustrates an initial manual approach to providing an implementation. It defines an implementation framework and a technique for implementing model translators based on their specification using the UML/OCL technique. The implementation approach described in this chapter provides a basis for the automatic approach described in the following chapter.

5.1 Introduction

A model translator specification is assumed to contain the specification of three parts:

1. the definition of the *two* models, in UML; and
2. the definition of the mapping relationships and associated constraints between the models and component parts.

The chapter discusses an overall implementation architecture and discusses the generic behavioural characteristics of the model and translation components.

The primary aim of any translator implementation is to keep the specified constraints valid; this is achieved by altering one or other model so that each constraint evaluates to true.

A secondary objective of the translator implementation is that the implementation of the models themselves are unaffected (or at least minimally affected) by the translation components. That is to say, that the implementation of the models should be independent of whether or not they are to be translated.

To achieve this two implementation approaches are discussed; one based on the Visitor pattern and one that uses the Observer pattern[8]. Both of these approaches require limited additions to the model components, which may be needed anyway as part of the implementation of other requirements in the application as a whole.

The Visitor based approach has been successfully used to implement the translations forming part of the Permabase project. The approach is straightforward but is a one step process and requires two translators to be built – one for translating in either direction. The major drawback of this approach, from the Permabase perspective, is that any change to the source model requires the whole translation process to be re-executed.

[8] These patterns are described in Chapter 2.

This problem led to the development of the Observer based approach. The approach is more complex to implement, but the translation process in each direction can be combined into a single set of classes. These classes “actively” translate between the models as incremental changes are made to either one. The implementation of the mappings and constraints must take account of how one model changes and alter the other model with respect to those changes.

The Java programming language has been chosen as the target language for implementing the translators. Segments of Java code and UML diagrams are used to illustrate the examples.

The rest of this chapter is organised as follows:

Section 5.2 describes the visitor based implementation approach. This approach is of particular interest with respect to a comparison with the observer-based implementation. The last subsection discusses issues and problems related to this implementation technique.

Section 5.3 describes the observer based implementation approach. The approach is based on a multi-way observer pattern that extends the basic observer pattern; support for this extension is described. The last subsection discusses issues and problems related to this implementation technique.

Both sections 5.2 and 5.3 illustrate the implementation using the Directed Graph to Tree translator that has been specified in the last chapter.

Section 5.4 concludes the chapter, discussing the advantages of each implementation approach. The section shows how the observer-based implementation meets the second of the objectives set out in the thesis introduction (chapter 1, section 1.2).

5.2 Visitor/Builder Implementation

The first implementation approach presented is the one used within the Permabase project. The technique implements a one step and one direction translator, using a Visitor pattern to traverse the source model and progressively build-up an appropriate target model.

5.2.1 Architecture

A visitor or target model generator class is developed that contains the translator implementation. Each visit method (of the Visitor class) corresponding to a source component constructs the appropriate target components. The built target components form part of a target model that is a valid translation of the source components, in accordance with the specified mapping constraints.

A ‘Builder’ interface is used to create the target model, allowing the same translator to build different implementation versions of the target model. This de-couples the building instructions from the actual target model being built.

Figure 49 shows the architecture of the translator. The Target Generator implements the Source Visitor interface and traverses the source model. The visit methods of the generator call build methods on a Target Builder to construct the translated target model. The link between generator and builder is via an interface to Target Builders enabling the substitution of alternative, specific, target model builders.

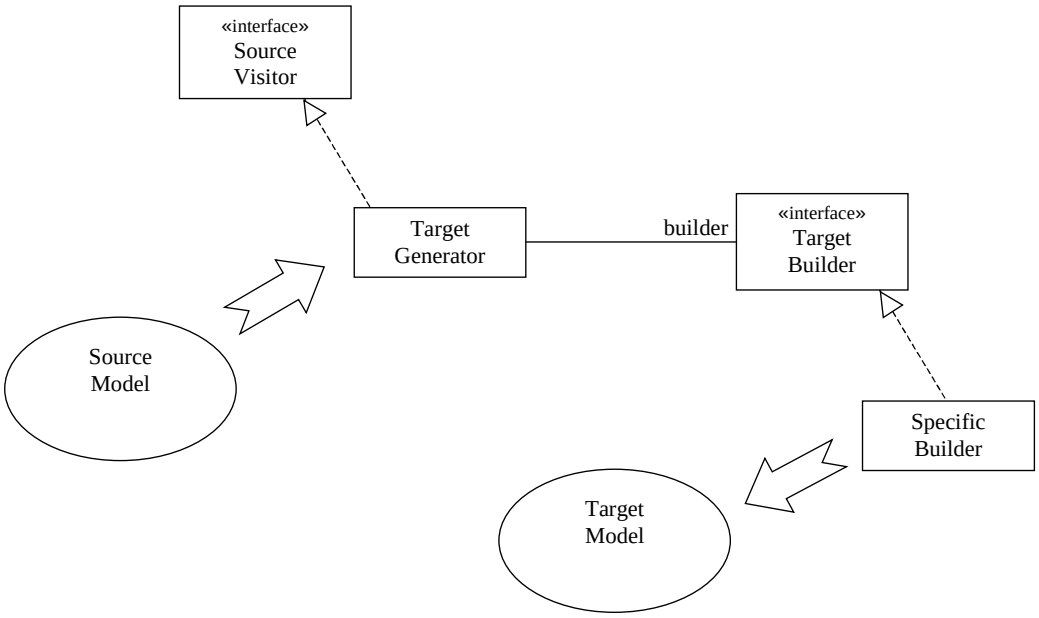

Figure 49 – Visitor/Builder Translator Architecture

The Target Generator (translator) is invoked by causing the source model to accept it as a Visitor. Specifically, the 'accept' method (part of the Visitable interface) is invoked on the root of the source model, passing the Target Generator as the Visitor parameter[9].

The source model is subsequently traversed in accordance with the traversal pattern as defined by the implementation of the visitor pattern. This causes each component of the source model to be 'visited' and thus the appropriate components of the target model are built.

5.2.2 Requirements of the Model Implementations

This architecture imposes three requirements on the models involved in the translation. In fact, the translator is unidirectional and the requirements are imposed, respectively, on the source and target models. If two translators are built, one for each direction, then of course the requirements are imposed on both models.

These three requirements are as follows:

1. The Visitable interface must be implemented by each component of the model, enabling the visitor to traverse the model. This involves implementing, for each component, an "accept" method that defines any traversal steps originating from the component and calls the visitor's visit method for that component.
2. A model specific visitor interface must be defined, specifying the visit method signature corresponding to each component of the model.
3. A target model builder interface must be defined, and a target model specific builder implemented.

5.2.3 Issues

There are two important issues to consider with respect to the Visitor based implementation technique. These are discussed in the subsections below:

[9] The Visitor pattern and components (Visitable interface, accept method and Visitor interface) are discussed in Chapter 2.

5.2.3.1 Reporting Translation Errors

Subsection 5.2.4.2 (below) indicates a situation in which one model cannot be translated, in accordance with the constraints, into the other; i.e. the DirectedGraph cannot be translated into a Tree.

The translator specification does not specify how to handle such a situation. Depending on the purpose of the translator, the condition may need to be handled in differing ways. A simple approach would be to simply output an error message, leaving it up to the user of the application in which the translation is involved to fix the problem. Alternatively, an exception can be thrown and caught by some part of the application that can either fix the problem, or give feed back to the user regarding the nature of the translation problem.

In either of these cases, the translation could either: terminate as soon as the problem is detected; or unwind the model traversal to a point where the translation can continue.

5.2.3.2 The Traversal Order

This leads to the second important issue, the model traversal order. The translation is performed as the Visitor traverses over the source model; thus, the definition of the target model creation code must take into consideration the traversal order, and hence the order in which it is called. Failure to do so can result in partially connected target models that are not valid translations.

5.2.4 Example

This example describes the implementation of a Directed Graph to Tree translator and the opposing Tree to Directed Graph translator based on the specification of these translators defined in the previous chapter. To recap, the UML/OCL based specification is shown in Figure 50.

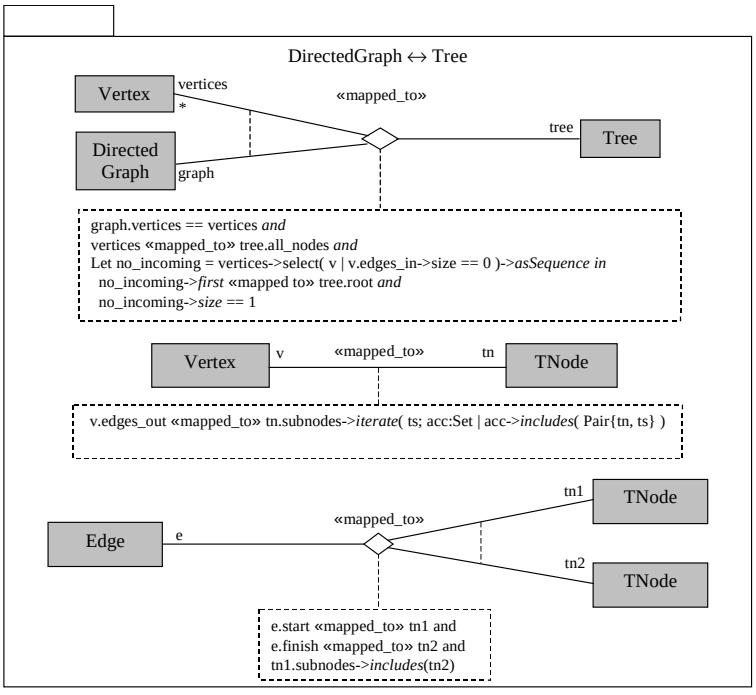

Figure 50 – DirectedGraph ↔ Tree Translator Specification

A Tree can always be converted into a Directed Graph; there are no difficulties in making such a translation, and hence this translator is presented before the Directed Graph to Tree translator. However, Directed Graphs can be produced that do not map to valid Trees; this translation introduces the need to handle such invalid cases within the translation process.

Both the graph and the tree must be traversed by their visitors; the traversal order must be defined as the translators depend on it. Trees can be traversed in a variety of orders and graph traversal could occur in a number of different patterns.

For this implementation, the Tree is traversed using a post-order pattern, and the graph visitor is defined to visit, in turn, each vertex followed by a visit to each edge.

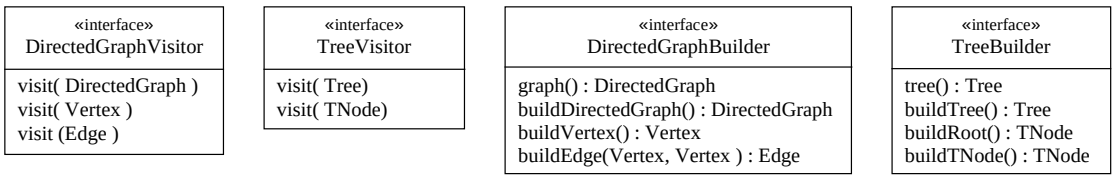

Figure 51 – Visitor and Builder Interfaces

The visitor and builder interfaces are shown in Figure 51. The visitors can be seen to define a visit method for each component of their models and the builders have methods enabling the models to be constructed and retrieved.

The traversal order for the DirectedGraph and Tree models is defined as shown in Figure 52 and Figure 53 below.

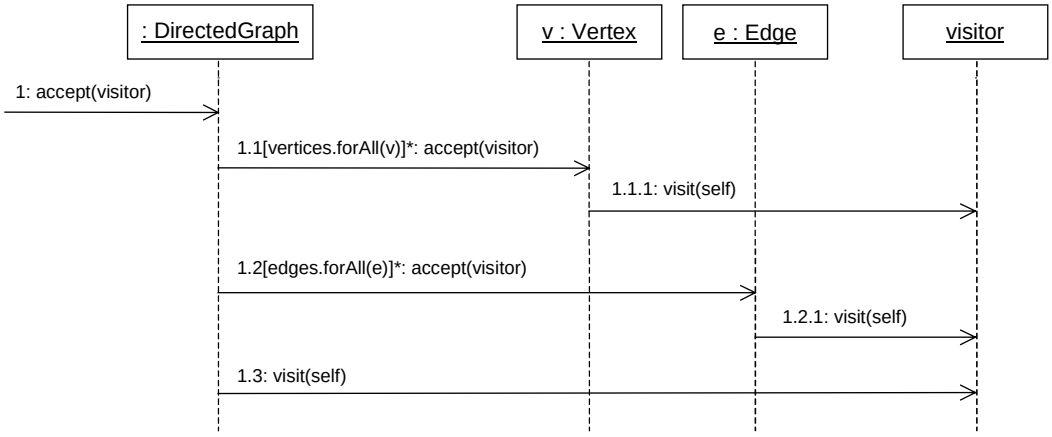

Figure 52 – Traversal Order for DirectedGraph Visitor

The traversal sequence of a DirectedGraph, starting from the containing DirectedGraph object, first visits every Vertex, then visits every Edge and finally visits itself.

The traversal sequence for a Tree, starting with the containing Tree object, first visits the root TNode, this visits each subnode before visiting itself, each subnode (also TNodes) visits its subnodes followed by itself. Finally, the Tree object visits itself. This is a post-order style of tree traversal – the subnodes of a TNode are visited *before* the TNode itself. The order of visiting the subnodes is not defined.

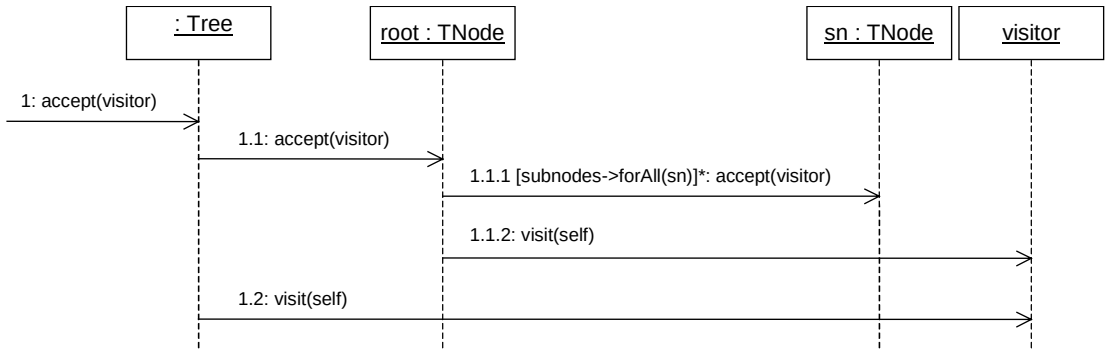

Figure 53 – Traversal Order for Tree Visitor

5.2.4.1 Tree → DirectedGraph

The Tree to DirectedGraph translator (or DirectedGraphGenerator) is implemented from the specification as follows:

1. There are two components visited by the tree visitor – the Tree and the TNode. According to the post-order visiting pattern, TNodes are visited first; hence, we start by considering this component.
2. The specification constraints, associated with a TNode, define:
    a) that a Vertex must exist for it to be mapped to, and
    b) that this node and each subnode is mapped to two Vertices and an Edge, where the second vertex is the vertex mapped to the subnode.
3. Specification constraints associated with a Tree define simply that the root TNode must be mapped to a Vertex.

This information deduced from the specification enables the implementation of the DirectedGraphGenerator to be as described in Table 6.

```
class DirectedGraphGenerator
  implements TreeVisitor
{
  public DirectedGraphGenerator( DirectedGraphBuilder b ) {
    builder = b;
  }
  private DirectedGraphBuilder builder;
  private Map mappings = new HashMap();
  void visit( Tree tree ) {
    // nothing
  }
  void visit( TNode tn ) {
    Vertex v1 = builder.buildVertex();
    mappings.put(tn, v);
    Iterator i = tn.subnodes.iterator();
    while (i.hasNext()) {
      TNode ts = (TNode)i.next();
      Vertex v2 = mappings.get(ts);
      builder.buildEdge(v1,v2);
    }
  }
}
```

Table 6 – Implementation of the Tree to DirectedGraph Translator

To create a DirectedGraphGenerator object, it must be passed a DirectedGraphBuilder object that is used to construct the graph. Visiting a Tree object doesn't require any actions for modifying the mapped DirectedGraph; visiting the TNode implements all of the translation process. A Vertex is created and recorded as the translation of the visited TNode. Subsequently, the subnodes are iterated over;

the vertex translated from each subnode is retrieved (the post-order traversal should ensure that one exists), and an edge is created from the vertex for the visited TNode to the retrieved one.

5.2.4.2 DirectedGraph → Tree

The implementation of the DirectedGraph to Tree translator is slightly more complex as it has to handle cases where the translation is invalid. Some parts of a graph cannot be mapped to equivalent parts of the translated Tree structure. For example:

1. A vertex without any incoming edges is translated as the root of the tree, hence multiple vertices with no incoming edges cause multiple roots. There can only be one root in a tree (although a Forest model could have multiple roots).
2. An incoming edge of a Vertex is used to determine the translated TNode's parent TNode. Edges that cause a vertex to have more than one incoming edge would cause a TNode to have multiple parents.

The traversal order for the directed graph visitor is Vertices, Edges, DirectedGraph; hence, if we consider the constraints in this order, the following information can be deduced:

1. Each vertex is mapped to a TNode.
2. An Edge defines the subnode/parent relationship between the two TNodes mapped to the Vertices at the ends of the Edge. If this relationship is already defined for the TNode mapped to the 'finish' vertex then the edge is invalid with respect to this tree translation.
3. The TNode mapped to the first Vertex that has no incoming edges is defined to be the root of the Tree. Other Vertices without incoming edges are invalid with respect to this tree translation.

The implementation of a TreeGenerator, formed from this information is as shown in Table 7.

```
class TreeGenerator
  implements DirectedGraphVisitor
{
  public TreeGenerator( TreeBuilder b ) {
    builder = b
  }
  private TreeBuilder builder;
  private Map mappings = new HashMap();
  void visit( DirectedGraph graph ) {
    //Find a list of root_vertices with no incoming edges.
    Vertex rv = root_vertices.first();
    builder.tree.root = (TNode)mappings.get(rv);
    // Indicate that all other root_vertices are invalid.
    ...
  }
  void visit( Edge e ) {
    TNode tn = (TNode)mappings.get(e.start);
    TNode ts = (TNode)mappings.get(e.finish);
    if (ts.parent == null) {
      ts.parent = tn;
      tn.subnodes.add(ts);
    } else {
      // Indicate that edge is invalid.
    }
  }
  void visit( Vertex v ) {
    mappings.put(v, new TNode());
  }
```

Table 7 – Implementation of DirectedGraph to Tree Translator

To create a TreeGenerator object, it must be passed a TreeBuilder object for constructing the Tree.

Visiting the DirectedGraph determines which Vertex represents the root of the Tree, and indicates other vertices which cannot be mapped. Visiting a Vertex simply enables the creation of a TNode.

Visiting an Edge is the most interesting part of the translation. It is assumed that the visiting pattern has caused TNodes to be created for each vertex. The task, upon visiting an edge, is to specify the subnode/parent relationship between the two TNodes mapped to the start and end of the Edge. There is the possibility that the edge should be marked invalid and there are two options for dealing with this:

1. To indicate that all edges coming into a Vertex are invalid if there is more than one, or
2. To build the tree based on one of the edges (the first encountered) and indicate that the others are invalid.

This implementation employs the second of these approaches.

5.3 Observer based Implementation

The second implementation approach for creating a translator is based on the Observer pattern; or more specifically, it is based on a multi-way enhancement of the observer pattern. This approach has been developed specifically to solve the problems inherent in the Visitor based approach.

Specifically there are two aims of this implementation:

1. That the translation process should take minimal time to execute.
2. That the translator implementation should operate in both directions.

The following subsections describe the implementation approach and the components that have been developed in support of it. Following this, there are subsections which address implementation issues such as Event Storms and Concurrency.

5.3.1 Architecture

The architecture for the observer based translator implementation is effectively a repeated application of the multi-way observer pattern (discussed below), connecting components from one model to components from the other; this is illustrated in Figure 54.

The translation process is divided up into “mini” translators that focus on translating between the minimum possible number of components. In some cases, this is between two components, one from each model. In situations that are more complex, the mini translators control the relationship between a few components from one model and one or more from the other.

The collection of all of these mini translators (or mapping components) forms the translator as a whole. As components are added to one model, new mapping components are created that relate these new model components to their counterparts in the other model (creating them if necessary).

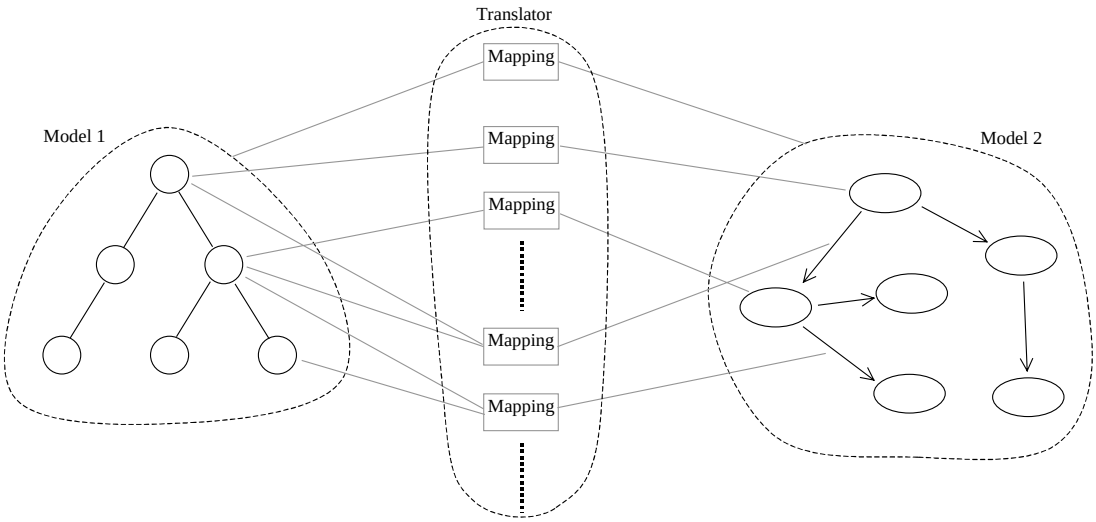

Figure 54 – Architecture for an Observer Based Translator Implementation

5.3.2 Managing the Mappings

Many mappings are created and combined to form the model translator as a whole. The management of these mappings is an essential part of the implementation.

A possible approach is to create a Mapping hierarchy, where each mapping notionally contains other mapping, for which it is responsible. However, the problem with this approach is determining the structure of the composition hierarchy. Copying the composition structure of either of the two models is not appropriate as each model may have a vastly different structure and the mappings may cross composition relationships.

An alternative is to provide a general mapping manager for the translator as a whole. This object contains and has responsibility for all of the mappings involved in the translation system.

All mappings are ‘registered’ with the manager, which can be subsequently interrogated to determine, for any model component or set of components, the opposing components from the other model.

A generic interface for this mapping manager is shown in Figure 55.

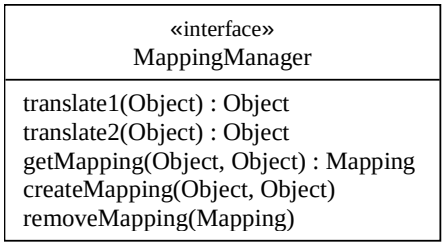

Figure 55 – UML definition of a generic MappingManager

A manager has two ‘translate’ methods; these methods are passed an object from one model and return the appropriate object from the other model, onto which the first object is mapped. If the mapping relates multiple objects from one or other model, they are grouped into a single tuple object.

A ‘getMapping’ method is used to access a specific mapping that exists between its two object parameters. A ‘createMapping’ method is provided for generating the

required mapping objects; this method should be overloaded for specific translators, defining parameters of the appropriate type for creating the respective mappings. The ‘createMapping’ methods are a variation of the Builder pattern and are used to create the mapping objects that collectively form the translator.

For complex translator implementations, it may be necessary to extend the ‘translate’ methods to enable them to distinguish between objects that may be involved in multiple mappings.

5.3.2.1 Relationships between Mappings

Within complex translator specifications, it may be possible to define relationships between different mappings. For example, model components are often related by a generalisation relationship; this could be reflected within the set of mapping specifications.

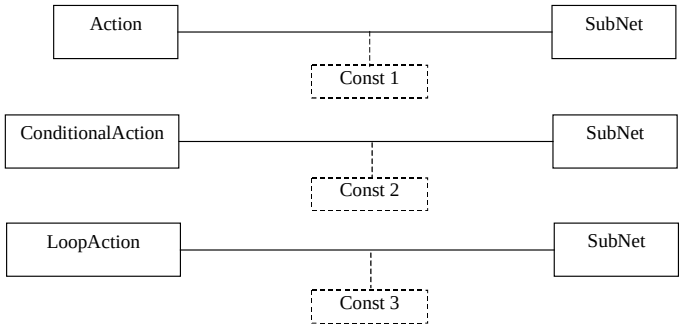

Figure 56 – Example set of Mappings

Figure 56 shows an example set of specifications that could form part of a translator specification between a UML style model of behaviour and a Petri-Net model, i.e. as discussed at the beginning of Chapter 4.

The components ConditionalAction and LoopAction are both subtypes of Action. Ideally, the translator specifications should be interpreted such that the constraints (Const 1) from the Action mapping are applied to the mappings for ConditionalAction and LoopAction along with the constraints specific to those mappings.

Other relationships could be defined between mappings, possibly defining constraints on the mappings themselves. For example, the application of a constraint to a mapping for efficiency; a mapping class could be constrained rather than defining a mapping between multiple objects. An example of this can be seen in the DirectedGraph↔Tree translator, as explained below.

The mapping specification between the graph and tree requires the inclusion of every Vertex of the graph so that one can be picked for mapping to the root of the tree. A possible alternative would be to relate the DirectedGraph↔Tree mapping and to each Vertex↔TNode mapping and constrain this relationship to define the relationship between one of the Vertices and the root TNode of the Tree.

These relationships between mappings are feasible from an implementation perspective, but the specification technique does not currently include a mechanism for specifying them. For this reason they are not currently used as part of the implementation technique, as the implementation is intended to be derivable directly

from the specification. Ideally, the specification technique should be extended to enable these specifications.

5.3.3 Requirements of the Model implementations

A translator specification is declarative and the mappings state whether one model is a valid mapping of the other. From an implementation perspective, it is not sufficient to simply know whether two models are validly mapped, rather it is necessary for changes to one model to be reflected in the other, so that they stay validly mapped.

To achieve this an event notification variation of the "Observer" programming pattern ([Gamma_etal_94], described in Chapter 2) is used. Each model component fires events every time some aspect of it is altered – such as adding or removing components to containers or by changing the value of attributes.

Each mapping is implemented as an observer of events from the groups of components it relates. Appropriate alterations are made upon receipt of an event so that the mapping constraints remain valid.

To support this implementation approach, three requirements must be met by the models:

1. The model components must support the event notification version of the Observer pattern.
2. Events must be fired that indicate changes to the models (including the type of event).
3. It must be possible to determine the source of an event.

5.3.4 Building Observable Models

The implementation of the translator components will be simpler if the models they translate implement the Observer pattern in a standard way. It is not necessary that they do; the two models could implement the pattern in completely different ways! However, for the purpose of this discussion a standard is desirable.

The majority of the standardisation is achievable by using a single library of supporting components (i.e. those described in Chapter 2), the differences however can occur in how changes are reported. Although the library defines the *ObservableEvent* class, the event description is an arbitrary String and there is nothing to enforce how or when the events are fired.

The following subsections discuss a standard set of event types that can be fired by an observable model. It also describes a standard way to implement models that fire these events and additionally describes some supporting components that aid the construction of observable models adhering to that standard.

5.3.5 Events

Object-oriented models are built out of interconnected objects. The information described by such a model is defined by the attribute values of the objects and the particular connections between them.

Each model component must fire events that indicate the changes to the structure of the model and changes to the attribute values of the model components. The possible

events are characterised into three types, based on the ways in which an object-oriented model is constructed.

The components of an object-oriented model (i.e. objects) are connected in two possible ways:

1. Attributes - the attributes of an object can be simple data values or links to other object. The links can be implemented either by value (containment) or by reference.
2. Collections – an object attribute can be defined as a collection, in which case the values of the collection can be either other objects or references to other objects.

Given these two mechanisms for structuring a model, three types of change can be made to it and consequently three types of event can be fired. These are:

1. Addition of an object to a collection;
2. Removing an object from a collection;
3. Changing the value of an objects attribute.

Given that these three types of event collectively describe the possible changes that can occur to a model, to aid the identification of an event, three additional event classes are sub-classed from the basic *ObservableEvent*. The definition of these event classes is shown in Figure 57.

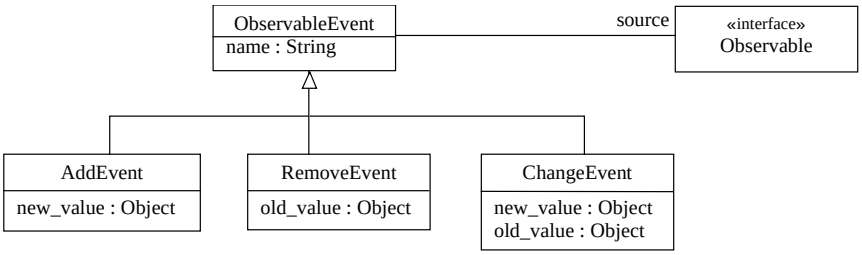

Figure 57 – Observable Events

The Add and Remove events carry a reference to the object added to or removed from a collection that changed. The *ChangeEvent* indicates both the old and new values of an attribute that has been changed. The name field of the *ObservableEvent* is used to define which, of the possibly many, attributes has changed; and the source link (attribute) defines the object (or collection) that initially fired the event.

5.3.5.1 Observable Components

When constructing an observable model component (i.e. its class specification), the implementation of behaviour for changing an attribute value or altering the content of a collection should be terminated with an action that fires an event indicating details of the change. If the component has methods that perform compound (multiple) changes to its attributes, then either multiple events must be fired indicating details of the change or a compound event must be fired that can be interpreted to give all the details of the changes.

It is customary to provide ‘accessor’ and ‘mutator’ methods for accessing attributes of objects; to meet the requirement of being observable the mutator methods should fire a *ChangEvent* after setting the attribute value.

Making collection attributes observable is slightly more complex. If accessor methods make the collection available, the collection itself must fire events to indicate the

changes. Alternatively, if the collection can only be manipulated by methods on the parent object (i.e. add and remove methods), these methods must indicate the changes.

To aid the construction of observable models, a number of component classes have been developed, which support the requirements of observable objects. These are wrapper classes around the basic data and collection types provided by the Java language and are illustrated in Figure 58.

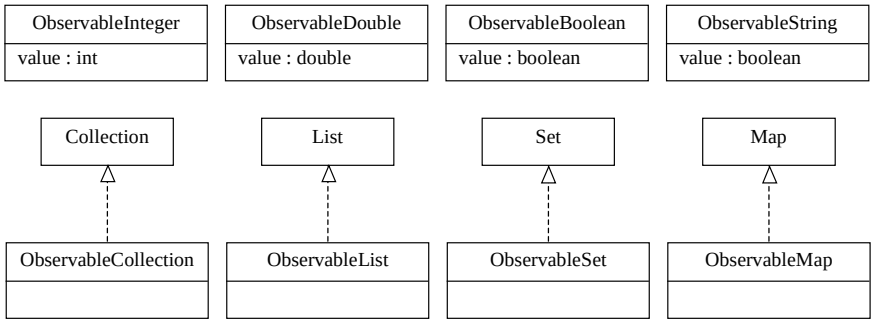

Figure 58 – Observable data and collection types

The wrappers support the generation of events that indicate changes to the components. The Integer, Double and Boolean classes fire 'Change' events when their values are changed. The Collection types fire 'Add' and 'Remove' events when components are added or removed from the collections.

Other model specific components must manage the firing of events explicitly according to the technique outlined at the beginning of this subsection.

5.3.6 Meeting the Requirements imposed on the Models

The basic requirements of an observable model (as set out in subsection 5.3.2) are supported as follows:

1. The Observable functionality is supported by all model components (if the proposed standard is followed and the library used). Bespoke objects must fire change events when their attribute values are set (or changed).
2. Changes to a model are indicated by the type of the event fired and the new/old values carried by the event. Three possible events are defined:
    a) AddEvents – when components are added to a collection
    b) RemoveEvents – when components are removed from a collection
    c) ChangeEvents – whenever the attributes of a component are changed.
3. Each of the above three event classes records the source object that fired the event and a name indicating which of its attributes has changed.

5.3.7 Multi-way observer pattern

Each mapping (or mini translator) is implemented as a separate object that listens for events from the model components it is relating. Each event signifies a change to the model component that fired it and the mapping must cause appropriate changes to the other components in order to cause the specification constraints to remain valid.

To implement a mapping object a "multi-way" extension to the standard observer pattern is used. Rather than the Observer simply listening to one Observable object, it listens to two (or more) different and distinguishable objects. Events from one object

are interpreted by the Observer and cause changes to another. This is illustrated in Figure 59.

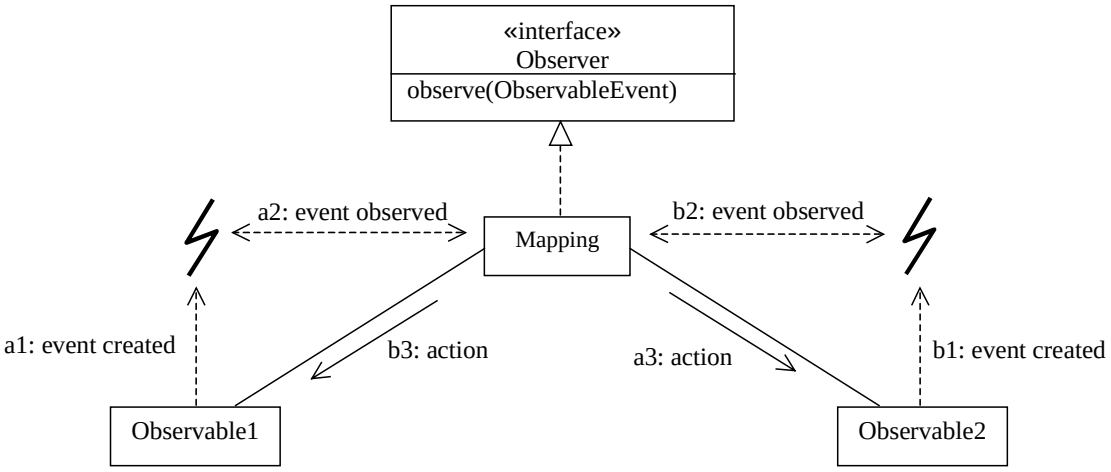

Figure 59 – Behaviour and Structure of a Mapping Object

Each mapping is at its core, an observer of two sources of events. Depending on the source and nature of the event, a different set of actions must be executed in order that the constraint remains valid.

A Mapping object is implemented as an Observer, observing events from the various model components involved in its mapping. Figure 60 illustrates the architecture of a mapping object taken from the DirectedGraph↔Tree example.

Although a mapping object may observe more than two objects, it is thought of as having two sides – one for the components from each model. The multiple components from each side can be wrapped up in tuple-groups to keep the two-sided notion.

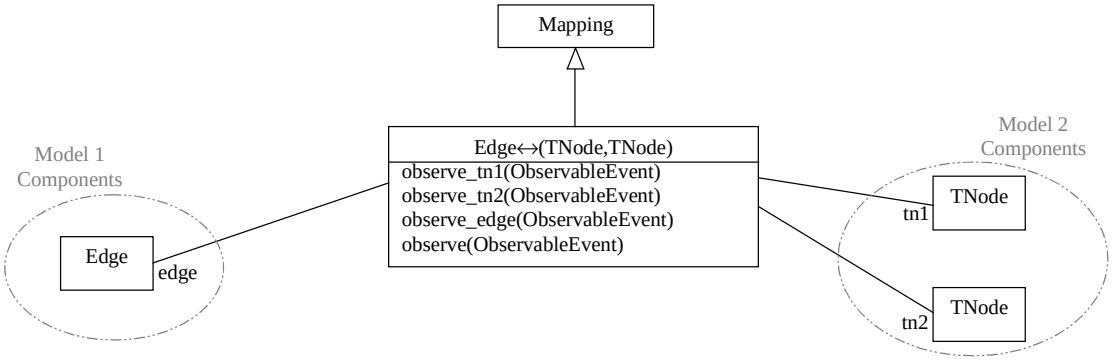

Figure 60 – Edge↔(TNode,TNode) Mapping Component

The *Edge↔(TNode,TNode)* mapping implements the *Observer* interface. It relates an *Edge* component from a *DirectedGraph* to two *TNode* components from a *Tree*; the *DirectedGraph* is denoted as Model 1 and the *Tree* as Model 2. This mapping implements the "observe" method by splitting it up into three sub methods, one for observing events from each of the pair of components of Model 2 and one for observing the events from the edge component of Model 1.

5.3.8 Implementation of the Mappings

The mappings between model components are specified using OCL constraints. The implementation however, attempts to keep these constraints valid by reacting to events caused by changes to either of the mapped components.

The implementation of the mappings is hence, by definition, specific to the mapped components. Even so, some general mechanisms can be defined and some generic support be provided. This section illustrates some example mapping implementations and describes generalised support for mapping class implementations, and management of the mappings.

Each mapping specification relates one or more components from one model to one or more components from the other. The constraints defined for the mapping specify the conditions under which the mapping is considered valid. Consequently, the implementation of the mapping must consider all events (changes to the model components) that may invalidate the mapping.

A starting assumption is made that the mapping is valid when it is first created. Thus, all subsequent, relevant, events may invalidate the mapping.

Events are generated for changes to each attribute of each of the model components involved in the mapping, however, some of these may be irrelevant to the mapping. The first step is to determine which events are of interest, by determining those that may invalidate the constraints.

The relevant events are deducible by analysing the content of the constraints. Any object attribute that forms part of the constraint, if changed, may invalidate the constraint by virtue of its value being changed.

It is at this point that the advantages of a standard event generation scheme are reaped. It is known that collection based attributes generate Add and Remove events, that single object attributes generate Change events, and the event name field will contain the name of the attribute altered.

So, the events to be monitored are those that are generated for each attribute involved in the constraints for each object involved in the mapping. For example, Figure 61 shows a hypothetical mapping specification and the attributes of the classes involved in it. (The grey elements in the figure are parts of each model that are relevant but not directly part of the mapping.)

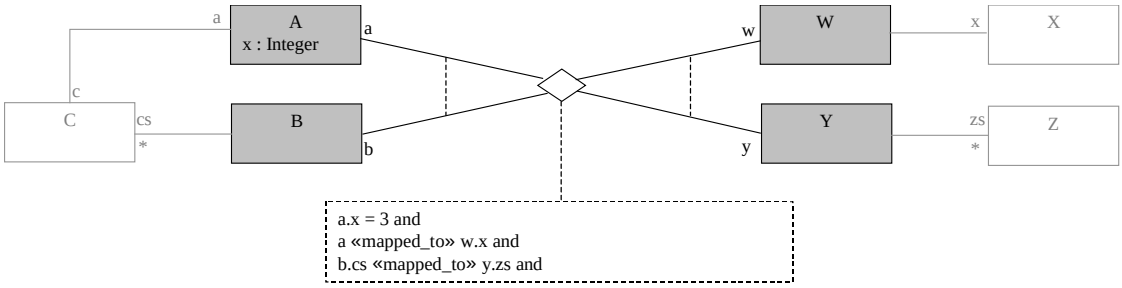

Figure 61 – Hypothetical mapping specification

Classes A,B,W and Y are involved in the mapping. A, W and Y have single object attributes and B has a collection attribute. The constraint involves the following attributes:

- a.x;
- w.x;
- b.cs; and
- y.zs

Thus the events that must be monitored are those which relate to these attributes. Table 8 shows the events that are generated for each object, and indicates which are relevant to the constraint.

Object	Attribute	Events generated	Relevant to constraint
a	x	Change	yes
a	c	Change	no
b	cs	Add, Remove	yes yes
w	x	Change	yes
y	zs	Change	yes
c	a	Not Observed	no

Table 8 – Events and their relevance to the constraint

Attributes ‘a.x’ and ‘w.x’ are involved in the constraint and create Change events, therefore they must be monitored. Attributes ‘a.c’ is not involved in the constraint; hence, it’s events can be ignored. Attributes ‘b.cs’ and ‘y.zs’ are collection attributes and therefore generate Add and Remove events, both of which must be monitored. Finally, attribute ‘c.a’ is not involved in the constraint, nor is it involved in the mapping so its events are not observed by this mapping object and cannot be monitored. If it were involved in the constraint, the mapping specification would be ‘ill-formed’ (see previous chapter) as its events cannot be monitored.

The implemented mapping object is set-up as an observer of each of the objects involved in the mapping. The appropriate ‘observe’ methods of the mapping determine the source of a particular event and can further deduce the attribute that caused the event, as this information is passed as part of the Event description. Thus, the monitoring of each of the possible events, from each relevant attribute, of each object, can be implemented as a separate sub-‘observe’ method. Table 9 illustrates a template for the implementation code based on this analysis.

```
class AB_WY
  implements IObserver
{
...
  public void observe_a_x(ChangeEvent ev) {...}
  public void observe_b_cs(AddEvent ev) {...}
  public void observe_b_cs(RemoveEvent ev) {...}
  public void observe_w_x(ChangeEvent ev) {...}
  public void observe_y_zs(AddEvent ev) {...}
  public void observe_y_zs(RemoveEvent ev) {...}
  //Observer Methods
  public void observeA(ObservableEvent e) {
    if (e.name().equals("x")) observe_a_x((ChangeEvent)e);
  }
  public void observeB(ObservableEvent e) {
```

```
    if (e.name().equals("cs")) {
      if (e instanceof AddEvent) observe_b_cs((AddEvent)e);
      if (e instanceof RemoveEvent) observe_b_cs((RemoveEvent)e);
    }
  }

  public void observeW(ObservableEvent e) {
    if (e.name().equals("x")) observe_w_x((ChangeEvent)e);
  }

  public void observeY(ObservableEvent e) {
    if (e.name().equals("zs")) {
      if (e instanceof AddEvent) observe_y_zs((AddEvent)e);
      if (e instanceof RemoveEvent) observe_y_zs((RemoveEvent)e);
    }
  }
}
```

Table 9 – Implementation template for Figure 61 mapping specification

The basic template for the implementation enables the effect on the constraint due to changes to any one particular attribute to be considered in isolation. The implementation of the behaviour for monitoring the attribute changes and keeping the constraints valid is dependent on the nature of the constraint and cannot be generalised.

5.3.9 Issues to Consider

As with the Visitor-based implementation approach, there are a number of issues and problems that must be considered with respect to this Observer-based implementation technique. The following subsections discuss these issues.

5.3.9.1 Invalid Mappings / Reporting Translation Errors

Reporting the errors within the observer-based technique is complex. The mapping objects respond to changes in one or other of the mapped components, some of these changes may invalidate the constraints and some may re-validate them again. Simply reporting an error message at the time of inconsistency is not sufficient as the message may become out of date over time (i.e. because of an attribute change that re-validates the constraints).

A solution is suggested, whereby the mapped components are defined to support an interface that enables them to be validated or invalidated. Whenever an attribute change is detected that causes the constraints in a mapping to be invalid, the object to which the attribute belongs is set to “invalid”; this indicates the particular object that is causing the constraint to be invalidated. When, or if, the attribute changes again, the object can be re-set to valid if the constraint is subsequently validated by the change.

The particular mechanism used to report the validity or invalidity of the model objects involved in the translation is left to the specifics of the application involving the translator. For example, if a visual representation of the objects exist, valid and invalid objects could be distinguished by a difference in colour.

This approach can be extended by passing details of the constraint that has been invalidated along with the change of state of an object’s validity. This information can then be used (e.g. by a user of the application) to determine a future change that revalidates the object and constraint.

5.3.9.2 Transitional States

An issue to consider with respect to a translation system is how to handle transitional states in a model. It is often the situation that during the construction process of a particular model it will pass through a number of 'transitional states' that are never intended to be complete or consistent models. Such models may possibly not have a valid mapping into a translated target model.

When using the single step translation approach of the visitor-based implementation this is not a significant issue. If an attempt is made to translate a model that is in a transitional state, the translator will report errors – this is an expected situation, a solution to which is simply to not attempt a translation on such a model.

With respect to the observer-based implementation approach, the issue is less obvious. One cannot simply *not* translate a model – the main objective of the implementation approach is that the translations happen continuously as a result of each change to the model. Consequently, the attempt to translate a model (or model component) will always occur even if the model should be considered to be in a transitional state.

No additional implementation is necessary to handle these cases; the error reporting mechanism will simply cause the relevant objects to be invalidated until the model moves out of the transitional state, when the objects will be re-set to valid.

To avoid this movement of the objects through an invalid state during a sequence of model changes that go though transitional state, it is necessary to 'wrap up' the changes. The sequence of steps must be wrapped up in such a manner that the changes can be reported (via the observer/observable mechanism) as a single change; thus, the mapping objects will not observe the intermediate 'transitional' states.

This is similar to the idea of a 'transaction' found in many database implementations; the transaction is formed from a number of small changes and then committed as a whole. Such changes to the model must be indicated by firing multiple events or special types of compound event.

5.3.9.3 Event Storms / Avoiding Live-Lock

This problem is a side effect of providing the facility that updates the translation based on changes to either model, i.e. enabling the translation to be bi-directional.

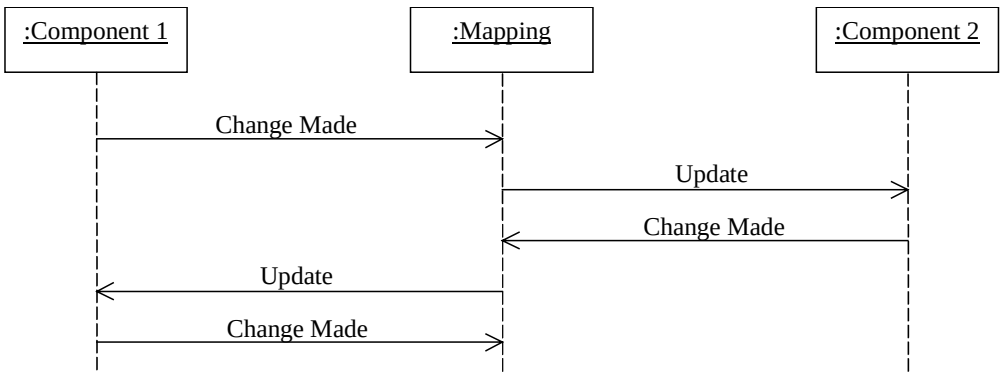

Figure 62 – Mapping Update Live-Lock

A mapping between two model components observes changes to each of the components and updates them according to the changes. A naïve implementation will result in a live-lock loop that repeatedly changes one component, observes the

change, updates the other component, observes this change… etc. The problem is illustrated in Figure 62.

One solution to this problem would be to provide two interfaces to the model components: one that changes the component and fires an observable event; and another that changes the component without firing the event. The latter of these could be used by the Mapping object to perform its updates and thus, the loop will not occur.

However, there are two drawbacks to this approach. Firstly, it requires that the model components have two different interfaces for essentially performing the same operations, this means a more complex implementation of the model components is needed, which is not desirable.

The other problem with this approach is that model components cannot take part in a chain of mappings. Many applications may have an architecture that causes a model to be involved in more than one translation, changes to one of the models are required to be fed through all of the translators, updating all of the models involved. If a mapping object updates a model component in such a way that no event is fired to indicate the change, then other mapping objects cannot observe the changes caused by the original update.

The solution to both of these issues is to stop event loops from occurring by altering the behaviour within the mapping objects. An update must be made to the target component, this component must also fire an observable event to indicate the change (in order that translators can be chained together). This event is necessarily picked up by the mapping object that makes the update, otherwise the bi-directional functionality of the translator is not implemented.

Thus, the event loop must be terminated in the mapping object. There is not any means to detect whether (from the perspective of a mapping object) an incoming event is as the result of an update made by this object or as the result of a change made to the observed component from another source. However, in many situations a test can be made to determine whether an update is necessary.

Take for example a mapping between two String attributes. A mapping object is defined to ensure that each string attribute contains the same string value. A change to the value of one of the string attributes required that the other be updated by setting it to the same value. The event loop can be halted in this situation by only making the update, if the target string attribute does not already contain a value equal to the source string attribute. This example is illustrated by the code in Table 10.

```
observe_object1_name( ChangeEvent ce) {
  String new_txt = (String)ce.new_value();
  if ( ! object2.name().equals(new_txt) ) {
    object2.name().setTo(new_txt);
  }
}
```

Table 10 – Event Loop Safe Implementation of a String Attribute's Update Code

Another example is that of a mapping between two collections of objects. This type of mapping frequently occurs in various different forms. A generalisation of the mapping specification is that if an object exists in one of the collections, an equivalent object must exist in the other collection and these two objects must be mapped. Essentially this is an implementation of the version of the «mapped_to» operator that takes two Collections as its parameters (see Chapter 4).

In this situation, the newly added object cannot be compared with an object from the target collection; the mapped object of the newly added one may not exist yet! However, we can use this fact to stop the event loop. If a mapped object of the newly added one does exist, then this mapping is valid and no update is required. Alternatively, if there is no mapped object, then a new one must be created, mapped to the newly added object, and added to the target collection.

Note: It is important that the mapping between the newly created object and its counterpart be created before the counterpart object is added to the target collection. If not, the event loop cannot be stopped by testing for the existence of that mapping.

A different option is to disable the observation mechanism of a translator whilst it is making the update. This could be achieved by stopping it from observing the changed component completely, or by stopping the observation of certain events from the observed components.

For instance, a Mapping between two object attributes can be implemented as shown in Table 11.

```
boolean _observe_object1_att = true;
observe_object1_att( IObservableEvent e ) {
  if (_observe_object1_att) {
    ...
    _observe_object2_att = false;
    update_object2_att();
    _observe_object2_att = true;
  }
}

boolean _observe_object2_att = true;
observe_object2_att( IObservableEvent e ) {
  if (_observe_object2_att) {
    ...
    _observe_object1_att = false;
    update_object1_att();
    _observe_object1_att = true;
  }
}
```

Table 11 – Alternative Event Loop Safe Implementation of example Update Code

The event loops are stopped by disabling the observe actions of this mapping object for the attribute(s) that are about to be altered. A simple boolean flag is used as the enable/disable mechanism.

This option is the more straightforward of the two, but introduces the need for extra state variables. It is also necessary to remember to set and reset the flags in the appropriate places.

More complex event loops involving multiple mapping objects can be conceptualised that are not stopped by either of these mechanisms. For example, a looped chain of mappings as shown in Figure 63.

There is currently no mechanism for stopping the event loop caused by this combination of mappings; however, a well-written specification should not contain such a definition. Tools supporting the evaluation of constraints can help detect these.

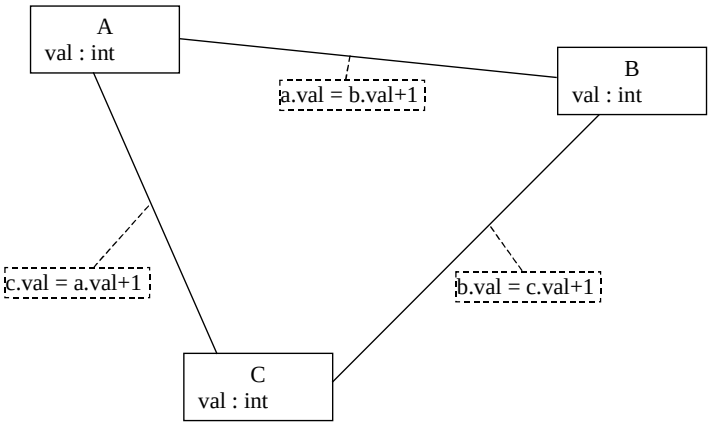

Figure 63 – A Looped Chain of Mappings

5.3.9.4 Concurrency

If multiple threads are used, care must be taken to ensure that the information in an event is still valid when it is observed. In a multi-threaded environment, it would be possible for multiple changes to occur to a model component before the actions in a mapping object, caused by observing those changes, are executed.

The solution to this is to pass, as part of the event notification, all the information required for performing the updates. In some cases, this may require a snapshot of a large portion of the event source's model.

5.3.9.5 Intra-Model Validity

Many models have constraints defined over them to ensure they are internally well formed. For example, the definition of a class inheritance model would define that no circular paths can exist in the inheritance hierarchy.

It should be noted that it is not the purpose of the mapping objects to enforce such intra-model constraints. It is assumed that some other agent controls issues related to those constraints. The mapping object could however respond to some attribute of a model component that indicates its internal constraints have been invalidated.

5.3.10 DirectedGraph↔Tree Example

This subsection describes the (manually generated) implementation of the first of the three mappings defined for the DirectedGraph↔Tree translator example specified in the previous chapter. Each of the implementations follows the template style discussed above.

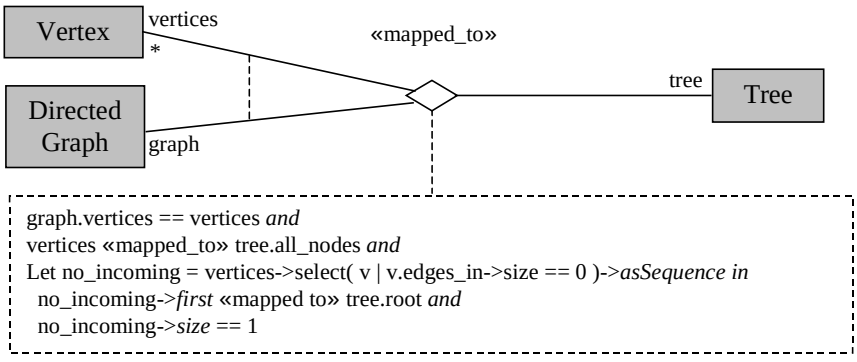

Figure 64 – DirectedGraph↔Tree mapping specification

The first mapping is the mapping between the DirectedGraph and Tree classes themselves. Figure 64 shows the specification of this mapping.

The constraints on the mapping (Figure 64) between a *DirectedGraph* and a *Tree* define that each Vertex in the graph must be mapped to a TNode in the tree. They also determine the mapping between the root of the *Tree* and a *Vertex* in the graph. Specifically, the constraint states that for the mapping to be valid:

- Every vertex in the graph must be mapped to every TNode;
- there must be a single Vertex in the graph that has no incoming edges; and
- this Vertex is mapped to the root of the Tree.

The constraint analysis is shown in Table 12. It shows that each component of the mapping is relevant to the constraint and must therefore be observed.

Object	Attribute	Events generated	Relevant to constraint
graph	vertices	Add, Remove	yes
graph	edges	Add, Remove	yes
tree	root	Change	yes
vertices (set)	edges_in	Add, Remove	yes

Table 12 – Analysis of Figure 64 constraints

This analysis gives the basic template for the mapping implementation (shown in Table 13). An ‘observe’ method is provided for each event produced by each attribute of each object that is relevant to the constraint.

```
class DirectedGraph_Tree
  implements IObserver
{
  ...
  public void observe_graph_vertices(AddEvent ev) { ... }
  public void observe_graph_vertices(RemoveEvent ev) { ... }
  public void observe_graph_edges(AddEvent ev) { ... }
  public void observe_graph_edges(RemoveEvent ev) { ... }
  public void observe_tree_root(ChangeEvent ev) { ... }
  public void observeAll_vertices_edges_in(AddEvent ce) { ... }
  public void observeAll_vertices_edges_in(RemoveEvent ce) { ... }
  ...
}
```

Table 13 – Implementation template for DirectedGraph↔Tree mapping class

The specific implementation of the observe methods for each attribute event is separately considered in the following subsections. Each part of the sub-expression must be examined to determine if the observed event has an effect on the validity of the constraint as a whole.

5.3.10.1 Adding a vertex to the graph

This event is caused whenever a new vertex is added to the directed graph as a whole. The constraint contains a number of sub-expressions, joined conjuctively. The implementation of each of these sub-expressions can be separately considered.

The first part (“graph.vertices == vertices”) states that the set of vertices monitored by the mapping is the same set as the set of vertices contained in the graph. This can be implemented by defining the method that returns the set of vertices involved in the mapping to return the set of vertices contained in the graph. This is shown below:

```
public Set vertices() { return graph().vertices(); }
```

The next part (“vertices «mapped_to» tree.all_nodes”) states that every vertex is mapped to a TNode . So, as the initial state must be that there are no vertices or they are all correctly mapped, every time a new vertex is added, a TNode must be created and mapped to it. The implementation of this is shown below:

```
IVertex v = (IVertex)event.new_value();
ITNode tn = new TNode();
tree().all_nodes().add(tn);
manager().createMapping(v,tn);
```

However, it is also essential to consider the possibility of a vertex being added to the graph that is already mapped to a TNode, i.e. a different event, from the Tree model may create the TNode and mapping. Consequently, a check must be added to protect this code segment from causing multiple mappings; as shown below:

```
if ( manager().translate1(v) == null )
  IVertex v = (IVertex)event.new_value();
  ITNode tn = new TNode();
  tree().all_nodes().add(tn);
  manager().createMapping(v,tn);
}
```

The ‘if’ statement checks for an existing mapping relating the new vertex, a new one is only created if necessary.

The third sub-expression is a Let statement that defines the set of vertices that don’t have incoming edges. The constraint uses a select function which (by the definition of OCL) can also be written as an iterate statement, i.e.:

```
vertices->iterate( v; acc:Set = Set {} |
 if (v.edges_in->size == 0) then
  acc->including(v)
 else
  acc
 endif )
```

Using this interpretation, the Let can be implemented as the method shown below:

```
public List no_incoming() {
  List acc = new Vector();
  Iterator i = vertices().iterator();
  while (i.hasNext()) {
    IVertex v = (IVertex)v.next();
    if (v.edges_in().size() == 0) {
      acc.add(v);
    }
  }
  return acc;
}
```

The next sub-expression (“no_incoming->first «mapped_to» tree.root”) determines which vertex should be mapped to the tree root. The vertex to map must be one with no incoming edges, hence if there is only one vertex in the whole graph that meets this requirement (the one just added) this must be the root of the tree; implemented as follows:

```
List no_incoming = no_incoming();
if ( no_incoming.size() > 0 ) {
  IVertex v = (IVertex) no_incoming.get(0);
  ITNode tn = (ITNode)manager().translate1(v);
```

```
        tree().setRoot(tn);
      }
```

The final part of the constraint ("no_incoming->size == 1") states that there must be only one vertex with no incoming edges; if there is more than one, then the constraint is invalid. There is nothing written in the mapping specification to indicate what should occur if the constraint is invalid (see discussion subsection 5.3.9.1).

Given that the constraint has been invalidated due to inconsistencies that have arisen between the two models, the best solution is to indicate which components of each model are the causes of the inconsistency. This is implemented by indicating that the component or components are invalid with respect to the mapping.

With respect to the DirectedGraph↔Tree mapping, there are two options to consider; if there is more than one vertex that is a candidate for being mapped to the root, either:

1. indicate that all of them are invalid; or
2. pick one of them and map this to the root, then indicate that all others are invalid.

The second option is chosen for the example implementation. As each vertex is added to the graph, it can be checked for incoming edges. If it has none and is not the vertex mapped to the tree root, then it must be marked as invalid; this translates to the following code segment:

```
      if ( ( no_incoming.contains(v) ) && v != no_incoming.get(0) ) {
        v.setInvalid();
      }
```

Table 14 shows the combination of the code segments that implement the actions to be carried out upon observation of a vertex being added to the graph.

```
public void observe_graph_vertices(AddEvent ev) {
  IVertex v = (IVertex)ev.new_value();

  // vertices <<mapped_to>> tree.all_nodes
  if ( manager().translate1(v) == null ) {
    ITNode tn = new TNode();
    ((Translator)manager()).createMapping(v,tn);
  }

  // no_incoming->first <<mapped_to>> tree.root
  List no_incoming = no_incoming();
  if ( no_incoming.size() > 0 ) {
    IVertex v1 = (IVertex) no_incoming.get(0);
    ((IValidateable)v1).setValid();
    ITNode tn = (ITNode)manager().translate1(v1);
    tree().setRoot(tn);
  }

  // no_incoming->size == 1
  if ( ( no_incoming.contains(v) ) && (v != no_incoming.get(0)) )
    ((IValidateable)v).setInvalid();
  }
}
```

Table 14 – Implementation of actions resulting from adding a vertex

Note that it is necessary to validate a vertex when it is mapped to the root node due to vertices being invalidated when applicable. It is also necessary to consider the validation and invalidation of vertices when the root is changed as a consequence of events from the Tree model (see later discussion).

5.3.10.2 Removing a vertex from the graph

To implement the actions for responding to observation of a vertex being removed from the graph, a similar process is adopted. Each of the sub-expressions of the constraint is separately considered.

Firstly, to ensure that each vertex is mapped to a TNode. If a vertex is removed, then the appropriate TNode must also be removed along with the mapping that relates them.

It should be noted that the models themselves are assumed to take responsibility for ensuring that they are internally consistent. I.e. removal of a vertex from a directed graph may require removal of edges in order to avoid dangling ends; it is assumed that this task would be carried out by some other agent. It is not the responsibility of the translator to ensure the validity of the models within themselves.

The second sub-expression requires that the first vertex found without any incoming edges be mapped to the tree root. The implication to this part of the constraint with respect to removing a vertex is if the vertex mapped to the root is removed. If this occurs, a new vertex must be chosen (from those with no incoming edges) to be the root vertex.

Finally, the third part regarding only one vertex without any incoming edges requires that any vertices without incoming edges, unless mapped to the root, must be marked as invalid.

This results in the implementation code shown in Table 15 below.

```
public void observe_graph_vertices(RemoveEvent ev) {
  IVertex v = (IVertex)ev.old_value();

  // vertices <<mapped_to>> tree.all_nodes
  ITNode tn = (ITNode)manager().translate1(v);
  manager().removeMapping(v,tn);
  if (tn.parent() != null) {
    tn.parent().subnodes().remove(tn);
  }
  if (tn == tree().root()) {

    // no_incoming->first <<mapped_to>> t.root
    Iterator i = no_incoming().iterator();
    if (i.hasNext()) {
      IVertex vs = (IVertex)i.next();
      ITNode ts = (ITNode)manger().translate1(vs);
      tree().setRoot( ts );
    }

    // no_incoming->size == 1
    while(i.hasNext()) {
      IVertex vs = (IVertex)i.next();
      ((IValidateable)vs).setInvalid();
    }
  }
}
```

Table 15 – Implementation of actions resulting from removing a vertex

5.3.10.3 Adding an edge to the graph

This event occurs to indicate that a new edge has been added to the directed graph. The first part of the constraint is unaffected and the Let expression is always handled by the no_incoming method.

The part of the constraint that refers to the mapping between Vertices and TNodes is unaffected by adding a new edge.

The part that determines which vertex should be mapped to the root of the tree is dependent on the number of edges that finish at a Vertex. Adding an edge may cause the Vertex currently mapped to the root to be no longer a valid candidate for that position.

If the actions to implement this re-selection of the root are implemented here, they will be carried out any time that a new edge is added to the graph. However, it may be the case that an existing edge has its start and finish vertices redefined. In this case, a new root may still need to be re-selected but as a new edge isn't added, the actions will not be executed under this observe method. Instead, the actions are defined under observation of an incoming edge being added to one of the vertices in the graph (see below). The actions in that observe method will still be executed when new edges are added, as they will be connected to a vertex. The consequence of this is that no actions are executed in relation to this part of the constraint.

Finally, the last part of the constraint relates to the number of candidates for the root vertex, this is unaffected by the addition of an edge. The validity of the joined vertices is handled by the Vertex↔TNode mapping and the validity of the 'root' of the joined sub-tree/graph is altered by the actions detecting the addition of an incoming edge to a vertex.

Thus, no actions need to be carried out, giving the implementation shown in Table 16.

```
public void observe_graph_edges(AddEvent ev) {
  // vertices <<mapped_to>> tree.all_nodes
  // nothing

  // no_incoming->first <<mapped_to>> t.root
  // nothing

  // no_incoming->size == 1
  // nothing
}
```

Table 16 – Implementation of actions resulting from adding an edge

5.3.10.4 Removing an edge from the graph

The removal of an edge from the graph requires nothing to be carried out by a DirectedGraph↔Tree mapping object.

As with the analysis regarding adding an edge, the some parts of the constraint are unaffected by the edges in the graph.

The part of the constraint related to mapping the tree root requires no action to be taken if an edge is removed. The only vertex of interest is the one mapped to the root of the tree. Removing an outgoing edge from this vertex doesn't change its status as a valid root; and if the vertex is mapped to the root, it shouldn't have any incoming edges anyway.

This implementation is shown in Table 17.

```
public void observe_graph_edges(RemoveEvent ev) {
  // vertices <<mapped_to>> tree.all_nodes
  // nothing

  // no_incoming->first <<mapped_to>> t.root
  // nothing

  // no_incoming->size == 1
  //nothing
}
```

Table 17 – Implementation of actions resulting from removing an edge

5.3.10.5 Adding an incoming Edge to a Vertex

The only parts of the constraint that are affected by adding edges to any of the vertices in the graph (involved in the mapping) are the Let expression and the selection of the Vertex mapped to the tree root.

The implementation of the Let expression is unaffected as it is calculated each time that it is used, i.e. each time the method is called.

If an incoming edge is added to the vertex that is mapped to the root of the tree, then a new tree root must be selected. The new vertex to be mapped to the root could be selected from the set of vertices formed from 'no_incoming', or it could be defined as the vertex that forms a valid root from the sub-tree/graph connected by the addition of the new edge.

The second option is chosen to form the implementation shown here. A possible problem to watch out for is the addition of a sub-graph with no valid root vertex. However, this is not an issue in this case due to the constraints on edges and pairs of TNodes that stop loops from forming. Any edge added that could cause a loop, would re-define the parent/subnode relationships of the mapped TNodes rather than causing a loop. Edges that attempt to define multiple conflicting parent/subnode relationships in the tree will be marked as invalid.

The implementation is shown in Table 14 below:

```
  public void observeAll_vertices_edges_in(AddEvent ae) {
    IVertex vertex = (IVertex)ae.source();

    // vertices <<mapped_to>> tree.all_nodes
    // nothing

    // no_incoming->first <<mapped_to>> t.root
    ITNode tn = (ITNode)manager().translate1(vertex);
    if (tn == tree().root()) {
      IVertex v = vertex;
      Set incoming = v.edges_in();
      while(!incoming.isEmpty()) {
        IEdge e2 = (IEdge)incoming.iterator().next();
        v = e2.start();
        incoming = v.edges_in();
      }
      ITNode new_root = (ITNode)manager().translate1(v);
      tree().setRoot(new_root);
      ((IValidateable)v).setValid();
    }

    // no_incoming->size == 1
    //nothing
}
```

Table 18 – Actions to execute when an Edge is Added to a Vertex

5.3.10.6 Removing an incoming Edge from a Vertex

If an incoming edge is removed from a vertex, the only part of the constraint affected is that relating to the number vertices with no incoming edges. The vertex mapped to the tree root will have no incoming edges, and hence can not have one removed, thus any other vertex that has it's last incoming edge removed must be defined as invalid – there should be only one vertex with no incoming edges.

The implementation of this is shown in Table 15 below:

```
  public void observeAll_vertices_edges_in(RemoveEvent re) {
    IVertex vertex = (IVertex)re.source();

    // vertices <<mapped_to>> tree.all_nodes
```

```
    // nothing

    // no_incoming->first <<mapped_to>> t.root
    // nothing

    // no_incoming->size == 1
    if (vertex.edges_in().size() == 0) {
      ((IValidateable)vertex).setInvalid();
    }
  }
```

Table 19 – Actions to execute when an Edge is Removed from a Vertex

5.3.10.7 Changing the root of the tree

The tree root is the only tree model attribute that is relevant to this constraint. Changing the TNode that represents the root of the tree affects the mapping constraint as follows:

Part of the constraint, defines that all TNodes must be mapped to a vertex in the graph. This implies an that the TNode mapped to the root must be mapped to a Vertex in the graph, if one does not exist it must be created.

Part of the constraint defines that the root must be mapped to a vertex with no incoming edges. Thus, an assumption would be that the resulting action must mark as invalid any edges that are incoming to the vertex mapped to the root. However, if we assume that an invalid tree cannot be defined, then a TNode that is mapped to a vertex with incoming edges must be a child of another TNode and therefore cannot be set to the root of the tree.

The final part of the constraint implies that the vertex that used to be mapped to the root must be marked as invalid if it has no incoming edges.

This results in the implementation shown in Table 20.

```
public void observe_tree_root(ChangeEvent ev) {
  ITNode tn = (ITNode)ev.new_value();
  IVertex v = (IVertex)manager().translate2(tn);
  ITNode tn_old = (ITNode)ev.old_value();
  IVertex v_old = (IVertex)manager().translate2(tn_old);

  // vertices <<mapped_to>> tree.all_nodes
  if (v == null) {
    v = new Vertex();
    ((Translator)manager()).createMapping(v,tn);
    graph().vertices().add(v);
  }

  // no_incoming->first <<mapped_to>> t.root
  //nothing

  // no_incoming->size == 1
  if (v_old != null) {
    if (v_old.edges_in().size()==0) {
      if (v_old != v) {
        ((IValidateable)v_old).setInvalid();
      }
    }
  }
}
```

Table 20 – Implementation of actions resulting from changing the root

5.4 Summary

This chapter has discussed two architectures and programming techniques for implementing model translators from the UML/OCL translator specifications described in the previous chapter.

Firstly, a single step, Visitor based, implementation approach was discussed. This approach traverses over the source model of the translation building an appropriately translated target model throughout the traversal. The requirements on the models have been shown minimal, being the implementation of a Visitable interface for each model component.

The second approach discussed how to implement an active translator based on the Observer pattern. It has illustrated the event notification requirements of models that are to be translated using this approach, showing that this is the only requirement of such models. This enables a clean and separate distinction to exist in the implementation between each of the translated models and between the models and the translator.

The next chapter describes an automated approach to generating translator implementations. The approach is based on the framework of observable components described in this chapter.

Chapter 6

Automation

A manual approach to generating translator implementations is all very well, but it is very time consuming. This chapter presents a method for automatically generating an implementation of an observer-based active translator and uses a selection of examples to illustrate the approach.

The automated approach imposes more structure on the implementation framework than the manual implementation, producing translators that are slightly less efficient to execute, but enabling faster and easier generation of them.

The automated implementation uses the philosophy adopted in, [Emmerich_96] and [Finkelstein_etal_94] of separating the consistency checks from the actions performed as a result of inconsistency. This is appropriate as our specifications only specify consistency constraints; UML doesn't yet have a suitable action language, so the actions are not generated. Although the mapped to ('↔') operator provides some indication of suitable actions.

A UML/OCL translator specification can be considered platform-independent. The automatic implementation generator is itself a translation process, from the specification, to a platform-specific implementation of the specified translator. In relation to this, this chapter answers two questions, as follows:

- *What is a possible implementation platform for translators?*
- *What is the mapping of the specification to the implementation platform?*

Section 6.1 describes the framework of the platform on to which the UML based translator specifications are mapped. An important module of this platform is the observable OCL library, described as part of that section.

Section 6.2 illustrates, by the use of examples, the mapping from a translator specification to an implementation based on the presented platform.

6.1 Java and OCL Based Translator Platform

In line with the manual approach, Java is used here as a basis for the automatically generated translator implementations. The principle of creating mapping objects between two observable models is retained, although the process of implementing the inconsistency detection is simplified by the introduction of a library supporting observable OCL expressions.

The library enables the definition of OCL expressions over a network of Java objects. The objects must conform to the observable pattern discussed in the previous chapter, which enables the expression to detect changes to those components that may alter the evaluation of the expression. Upon detection of such a change, the expression

itself fires an observable event indicating that the expressions value may have changed.

The first subsection discusses the general architecture of the automatically generated framework. This is followed by a description of the observable OCL library and a description of how it is implemented. The final subsection shows how the mapping objects make use of the OCL library to indicate inconsistencies and shows which parts of the framework are left empty, to be completed with actions for responding to inconsistencies.

6.1.1 Translator Framework

The general framework of the implementation (Figure 65) follows the architecture of the specification. Two Java packages are assumed to exist, each containing an observable implementation of the two models involved in the mapping. The relationship between the two models is implemented in a third package, called “mappings”.

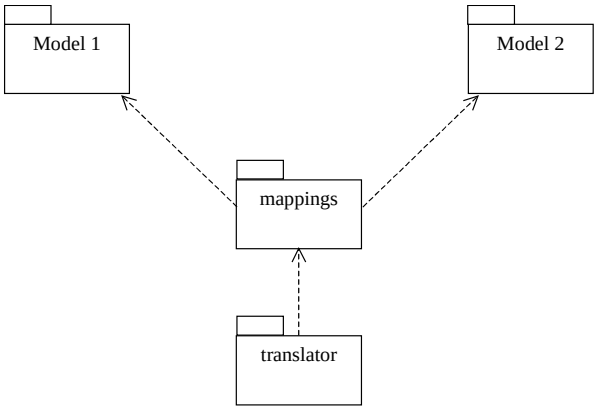

Figure 65 – The Automatic Translator Framework

The manual implementation did not separate the detection of an inconsistency from the required resulting actions; the code for both was included in the implementation of a single mapping object. For the automatic implementation, a distinction is made.

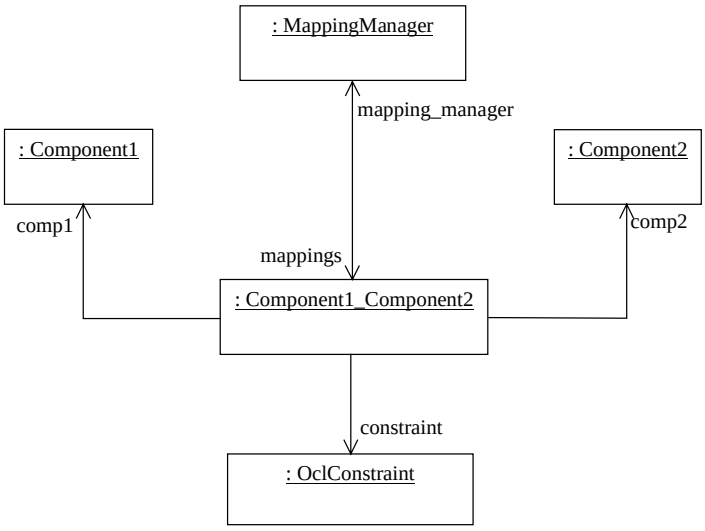

Figure 66 – An Example Mapping

Mapping objects are created that contain code that detects inconsistencies between the mapped objects in accordance with the specified OCL constraint. These are

controlled by a MappingManager, which enables creation of mappings and records the mappings created for a particular pair of models.

Figure 66 shows an example instance of the implementation of a mapping between two components. The mapping object contains a reference to each object (from the models) that is involved in this mapping and contains a reference to the MappingManager object that has recorded its existence. Additionally the mapping object contains an (observable) OclConstraint object that defines the appropriate consistency constraints.

The particular actions performed depend on the purpose to which the implementation is to be put. They may be required to simply report that an inconsistency exists, or (in the case of a translator) they may perform a sequence of changes that remove the inconsistency.

To enable reuse of the mapping implementation, the translator code is implemented as a separate package. The components of the translator package are as follows:

- **A Translator :** The translator controls overall management of the translation. The translation of a particular component is handled by either retrieving its existing translation from the consistency manager, or by creating a new translation using the appropriate generator.
- **A ConsistencyManager :** The consistency manager is an extension of the MappingManager defined in the mappings package. It controls creation of the consistency management objects and uses the mapping manager functionality to record their existence.
- **A Model1Generator and a Model2Generator :** These objects control creation of components from one model, depending on information about components from the other model; essentially performing the generative part of the translation. As part of the generation, they define the mappings between all components created; after generating the target translation of a particular source component, the source and target are assumed to be consistent according to the defined mapping constraints.
- **Several ConsistencyMapping classes :** Each mapping object defined in the mappings package is extended with a corresponding consistency mapping object defined as part of the translator package. The consistency mapping objects are intended to respond to events generated by the mapping constraint, and perform the appropriate actions. The particulars of these actions depend on the specifics of the translation.

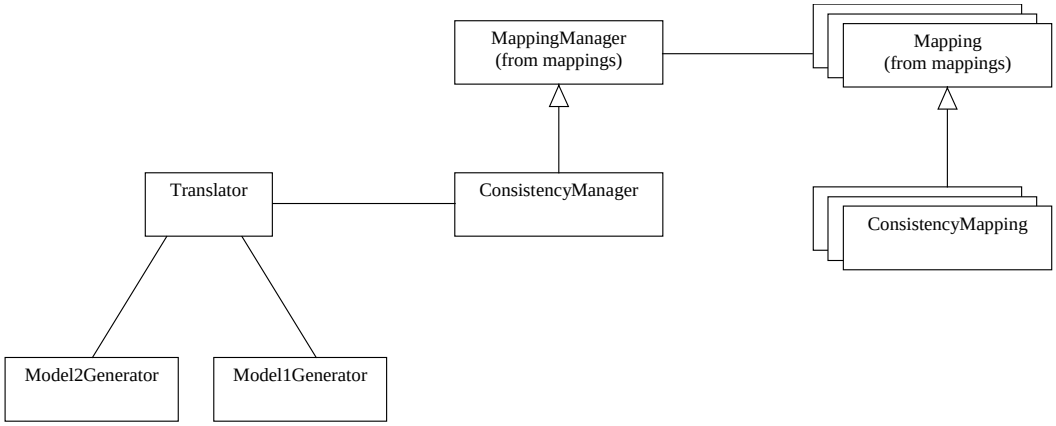

Figure 67 – General Architecture of a translator package

Figure 67 illustrates the general content of a translator package.

The next subsection describes the functionality and implementation of the observable OCL library used by the Mapping objects; it is essential to the easy automatic generation of these objects in a manner that they can provide events indicating inconsistency.

An automatic generator has been implemented in Java that generates an implementation of this framework. The generator takes as input, an XMI specification of the mapping specifications and produces as output a set of Java classes that implement the framework.

Future work is to specify this mapping using the UML/OCL specification technique itself and to use the automatic generator (over that specification) to bootstrap itself.

6.1.2 Observable OCL library

The observable OCL components provide a Java implementation of each of the OCL types defined by the standard. It provides an implementation of each basic type (Integer, Real, String, Boolean), of each collection type (Set, Sequence, Bag), and implements every operation defined on those types.

There are other libraries that provide this functionality (e.g. [Hussmann_etal_00]), however what is particularly useful about this implementation is that OCL expression formed using this library are observable. The standard types are extended to provide *mutable* versions, which can be altered (in the case of a StringBuffer) or have objects added to or removed from (in the case of a Collection). Changes to the mutable types are made detectable by implementing them as observable in the manner described in the previous chapter. Additionally, any Java class that is implemented in accordance with the observable functionality may also be used as a valid type within the OCL Expression.

An observable OclExpression class is defined, which represents a particular expression. This component fires events if any object involved in the expression is altered. This is achievable as each of the objects involved in the expression supports the observable functionality and the OclExpression object is defined to observe each object involved in the expression.

There are three primary features of the library:

1. A Java implementation of the OCL basic and collection types, supporting all the functionality defined in the standard.
2. A mechanism for specifying OCL expressions about a collection of Java objects.
3. The observable qualities of the OCL expressions.

The implementation of each of these features is discussed in the following subsections.

6.1.2.1 The OCL Types

The implementation of the OCL types is straightforward. Each OCL type defined in the standard is implemented as an interface defining each operation as a method (see

Figure 68[10]); infix operators, such as '+' or '/', are defined as methods using an appropriate name, i.e. 'add' or 'div'.

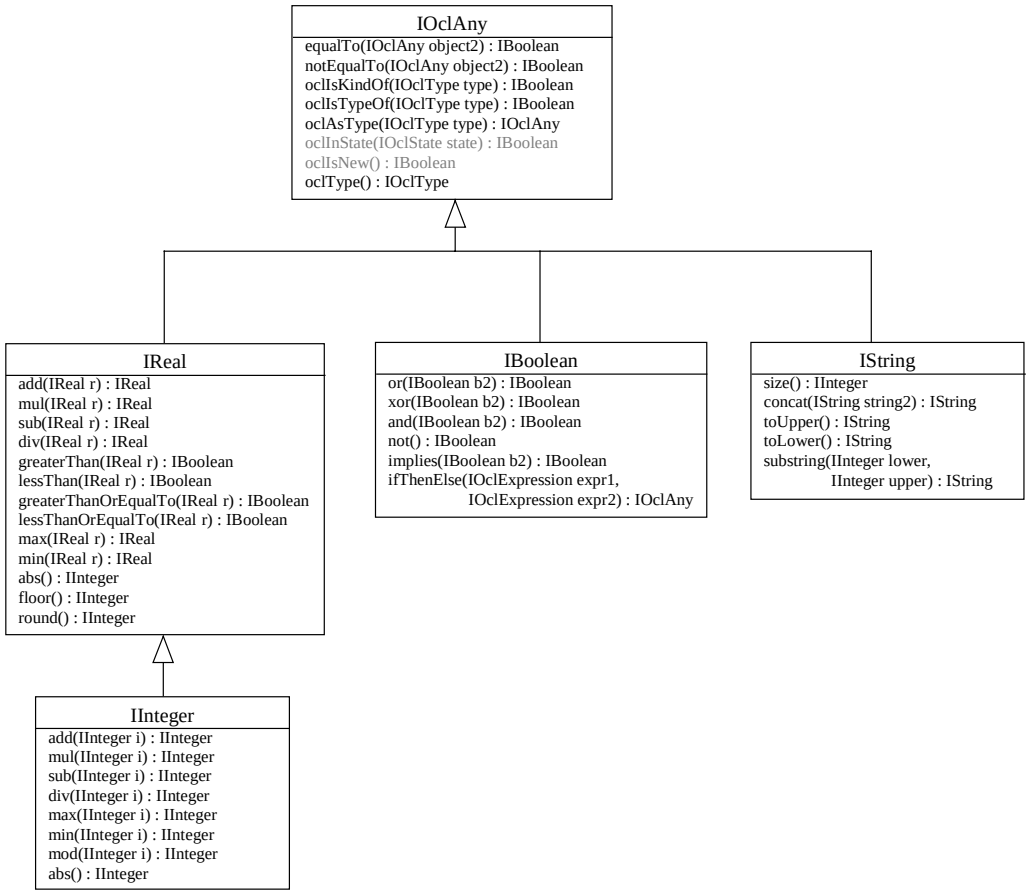

Figure 68 – Java Interfaces for the Basic OCL Types

Different implementations of the OCL interfaces can be provided; for example an observable or a non-observable implementation. Instantiation of the classes implementing the OCL types is carried out via a Factory class, whose methods construct the appropriate OCL object and return it as a type defined by the interfaces. Thus, all manipulations of the OCL objects can be carried out in terms of the defined interfaces, regardless of the particular implementation.

The OCL factory class must be implemented for each implementation of the OCL interface types. This class forms the interaction point (interface) between standard Java code and the OCL components. For example, the OCL Integer type is defined using an IInteger interface. The interface is implemented as a class named OclInteger, and the OCL factory class contains two relevant methods to aid the use of OCL Integers in the Java environment:

- `IInteger Integer(int i)`, which creates an OclInteger from a Java 'int' and returns the object as an IInteger.
- `int impl(IInteger i)`, which returns the Java 'int' implementation value of an OCL IInteger object.

Similar methods are provided for each of the OCL types; one to create instance of the OCL types from the Java versions; and one to create the Java types from the OCL versions.

[10] The light grey methods are currently not implemented by the library.

The implementation of the Collection types is also based on the Java equivalents. Similarly to the basic types, there are methods in the OCL factory class for creating Java or OCL collection types from each other.

Some operations require an OCL expression as a parameter; these can be passed an OclExpression object, as defined in the next subsection.

6.1.2.2 OCL Expressions

One aim of the library is to make the use of OCL expressions within the Java language as easy and seamless as possible. Towards this end, we require that OCL expressions can be entered as syntactically correct OCL text, whilst at the same time making use, within those expressions, of the methods defined in Java classes.

This subsection starts by describing how to incorporate OCL expressions within Java code and then described how the evaluation process of those expressions is implemented.

As with the creation of other OCL types, an OclExpression object is created using the OCL factory class. The actual text of an OCL expression is entered as a Java String; however, more information is required than simply the expression text. Each OCL expression is defined within a 'context'; this context provides names for the objects from which an OCL expression is constructed. It is necessary to provide each OclExpression object with the names, objects, and types of those objects; this provides its context.

The OCL factory method for creating OclExpression objects is defined as follows:

```
IOclExpression Expression(String expression,
                          Class[] types,
                          String[] names,
                          Object[] values)
```

Or there is a shorthand version for defining invariants:

```
IOclExpression Invariant(String expression, Object self)
```

This second method defines a single name "self" to be the name of the passed object, with the type being the class of the passed object. Each object within the expression must be navigated to starting with the name "self". An example of the use of these methods is shown in Table 21.

```
class X {
  IInteger _a;
  public IInteger a() {return _a;}

  IInteger _b;
  public IInteger b() {return _b;}

  IOclExpression inv1;
  IOclExpression inv2;

  public X( int a, int b ) {
    _a = OCL.Integer(a);
    _b = OCL.Integer(b);

    inv1 = OCL.Invariant("self.a = self.b", this);

    inv2 = OCL.Expression(a = b",
                          new Class[] {IInteger.class, IInteger.class},
                          new String[] {"a","b"},
                          new Object[] {this.a(), this.b()} );

  }
}
```

Table 21 – An example showing the use of OclExpressions in a Java class

In this example, the call to OCL.Invariant would be equivalent to the following call to OCL.Expression:

```
OCL.Expression( “self.a = self.b”,
                new Class[] {this.getClass()},
                new String[] {“self”},
                new Object[] {this} );
```

The implementation of an OclExpression makes use of the reflection capabilities of Java in conjunction with a library for creating Java classes at runtime [Dahm_99]. The call to OCL.Expression creates a new subclass of the OclExpression class and invokes an OCL parser over the expression. As the expression is parsed the body of a method, “evaluate”, is constructed on this new class; when completed, the constructed method contains a series of method calls that evaluate the parsed OCL expression.

The OclExpression subclass is constructed directly as Java byte-code as this avoids the necessity of having to compile it after creating it. As an example, the code in Table 22 shows some Java code that would compile into the same byte code constructed for the OCL Expression created for “inv1” from Table 21.

```
class OclExpression$1
  extends OclExpression
{
    OclExpression() {}

    public IOclAny evaluate() {
      return get("self").a().equalTo( get("self").b() );
    }
}
```

Table 22 – Java code illustrating a generated expression class

The ‘evaluate’ function returns the result of evaluating the expression. The series of method calls that form the evaluation are constructed by the parser as it analyses the original OCL text. The initial object for each navigation expression is retrieved using the ‘get’ method. The method is defined on the superclass ‘OclExrpression’ and it accesses the ‘context’ information passed to the OCL.Expression method, returning the object associated with the name given as the string parameter.

The type information passed as part of the context is used by the parser, in conjunction with the Java reflection functionality, to determine the methods to call on the initial objects. For instance, in our example:

- the parser would detect the need to call the method ‘a’ or ‘b’ on the object ‘self’;
- knowing that the type of the ‘self’ object is ‘X’, Java reflection is used to determine if there are methods called ‘a’ or ‘b’ on that class;
- if there are, then calls to those methods are added to the ‘evaluate’ method body.

If the OCL ‘feature calls’ require parameters, the same technique is used; the parser will analyse the types of the parameters and use reflection to determine if an appropriate method exists.

If the parameter type is ‘IOclExpression’ as is the case with many of the OCL Collection type’s operations then another Expression class is constructed to evaluate the expression passed as a parameter. The context of this new sub-expression is constructed from the context of the outer expression, adding any new names as required.

The next subsection describes how these expressions are extended to make them observable.

6.1.2.3 Observable Expressions

The ability to construct OCL expressions in the context of a set of Java objects is useful; however, in the context of this thesis, we are interested in when the evaluation of such an expression changes. It is not practical to repeatedly pole each expression and evaluate it; instead, the generated OclExpression classes are extended so that they become observable.

This enables an observer to be set up to respond to events fired by OclExpression objects. The expression objects are defined to fire an event anytime the expression changes value. To avoid evaluating the expression except when required, the observable expressions are defined to fire an event any time that any component of the expression changes and thus the observer can evaluate the expression if required. In order for the expressions to detect changes in their component objects, each object used in an expression must also be observable.

This functionality is implemented using the Observer and Observable interfaces described in Chapter 5. Each object used in an expression is assumed to implement the 'IObservable' interface, the observable expression observes these components and in turn fires an 'IOclExpressionChanged' event if a component change is detected.

To support the creation of Java models that are observable, the OCL library provides some extended versions of the OCL types that are both observable and mutable. Each of the collection types is extended by a mutable version and there is a mutable version of a String – a StringBuffer. These are as shown in Figure 69.

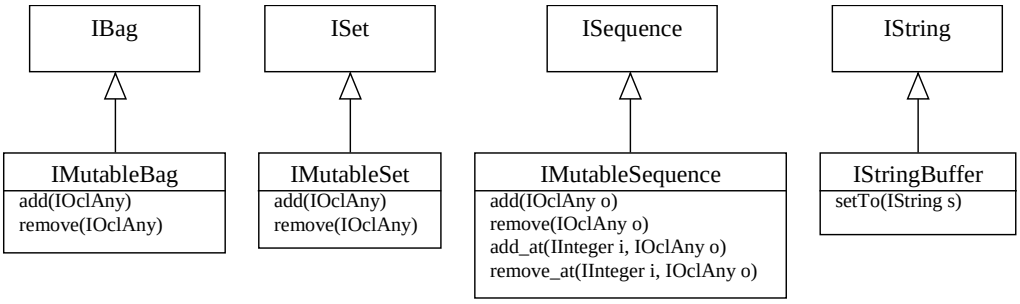

Figure 69 – Mutable OCL Types

Additional support is provided in terms of a set of Monitor classes. These classes are defined to observe OclExpressions and respond to OCLExpressionChanged events in different manners.

A base 'Monitor' class is defined to simply monitor an expression. Monitor objects can be used to set up 'watches' that call a specified method on a particular object upon receipt of a change event.

```
IOclExpression expr = OCL.Invariant(".......", this);
Monitor m = new Monitor();
m.watch(expr, "observe", this);
```

Table 23 – Example use of a Monitor object.

For example, the code in Table 23 defines a Monitor, that watches the expression 'expr' and when events are received indicating that the expression may have changed, the method '`observe(IOclExpressionChanged e)`' is called on the 'this' object.

Two subclasses of Monitor are provided – SystemOutMonitor and ExceptionMonitor. The first of these outputs the expression text and its evaluation to System.out whenever a change event is detected. The second causes an Exception to be raised if the evaluation of the expression is not the (OCL) Boolean value “true”.

Through using observable OclExpression objects, created using this library, the expression can be monitored for events that may change the evaluation of the expression. In the context of a consistency mapping for translator implementation, these events are monitored and actions can be defined that execute appropriate translating behaviour. The framework for this is described in the next subsection.

6.1.3 Implementation of Consistency Mapping Objects

A translator specification takes the form of consistency constraints and some consistency management actions. The UML/OCL mapping specification technique uses a non-standard OCL operator ‘↔’. The use of this operator gives added information regarding what the consistency management actions should be.

For example, given the specification of a mapping constraint between two objects:

obj1 ↔ obj2

The constraint is invalid if a mapping object does not exist between the two objects. One potential action, which would re-validate the constraint and achieve consistency, is to create the required mapping.

However, the automated approach does not currently interpret the OCL constraints to deduce the required actions. The automatic generator will create the classes required, but some of the content must be manually entered. Based on interpreting the semantics of the ‘↔’ operator, there is potential for much more automatic generation.

The purpose of the consistency mapping objects is to monitor the mapping constraint and take appropriate action when it is invalidated. The Monitor classes provided by the observable OCL library can help with this.

The general pattern for each consistency mapping class, is to create a monitor for the constraint and associate a function with each type of event and each possible source of event that can be generated by that constraint. These functions must subsequently specify actions appropriate to the required consistency management objectives. For example, reporting an inconsistency, or trying to alter the models to re-achieve consistency.

Figure 70 indicates a possible sequence of actions (method calls) generated by the framework upon detection of an event from a component in Model 1. Component1 causes an event indicating that attribute x has changed. This event is detected by the OCL constraint on the specific consistency-mapping object for Component1; the OCL expression simply detects that one of the components involved in the expression has changed. The actions to be performed upon detection of the change start in the consistency-mapping object. The first action must be to determine whether the change indicated by the event has altered the validity of the constraint; i.e. it is evaluated. If the constraint is invalid, appropriate actions must be invoked.

What those actions actually are, is dependent on the application of which the translator is a part. It may be that the constraint can be revalidated, or it may be that some feed back to the user is required to indicated that the change to Component1 has cause an invalid situation. In this case, it is assumed that the action required is to form

a translation for the new value for attribute x, hence a call is made to the translator requesting the translation.

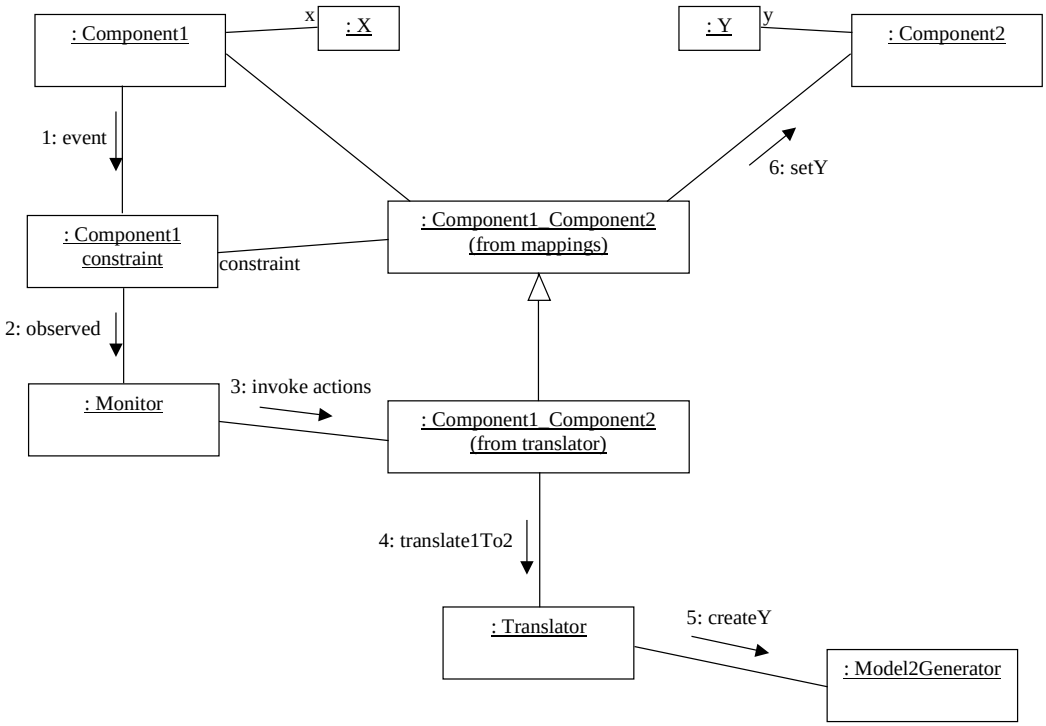

Figure 70 – Model Update Action Sequence w.r.t a Mapping Object

The translator determines whether a mapping for the new value already exists or not. If it doesn't, then a call is made to the generator for model 2 components, which requests that a new model2 component is created that is a valid mapping of the new value from model1. The generator is assumed to create the appropriate component (or components) and to request that a mapping be created between the components from each model. It is also assumed that the newly created model2 component(s) are consistent with the supplied model1 component, according to the defined constraints.

6.2 Examples

This section illustrates the use of the framework via the specification and generation of various different translator examples. Each of the following sections describes the problem to be tackled via the use of a translator, shows the UML/OCL specification of parts of the translator and describes the generation of the implementations.

The DirectedGraph↔Tree example used in the previous chapters, although appearing to be simple, does involve some complex issues. Consequently, we use two different examples to illustrate the process of generating the translator implementations. The issues with respect to the DirectedGraph↔Tree example are discussed in subsection 6.2.3.

The first example is used to illustrate the overall framework of classes created by the automatic generator. It also demonstrates which aspects of a translator implementation must be manually coded.

The second example illustrates how more complex one-to-many mappings are implemented and demonstrates how two different translators can be composed into a single application.

6.2.1 Java↔Tree

The first example illustrates a specification involving one-to-one mappings. The objective is to produce a ‘Tree view’ of the directories and files involved in a Java project. The project is assumed to consist of a number of directories, sub-directories and files; the requirement is to provide a view of this structure using the javax.swing.JTree component.

There are two aspects to providing this view, the definition and implementation of the two models and the definition and implementation of the translator.

Specification

The JTree component supports the use of any Tree style model, provided certain functionality is provided that enables the interface to the graphical part of the component. For the purposes of this discussion, that part of the implementation is not relevant. The specific Tree model used is the one used within the running example over the last two chapters and is shown in Figure 71. Note that each TNode has an additional ‘data’ attribute.

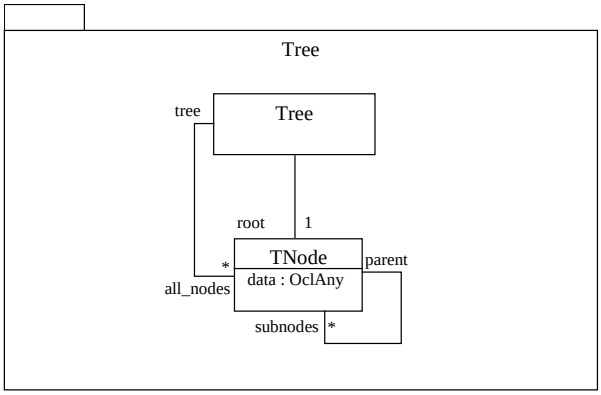

Figure 71 – The Tree Model.

A Model representing a Java project file and directory structure is shown in Figure 72.

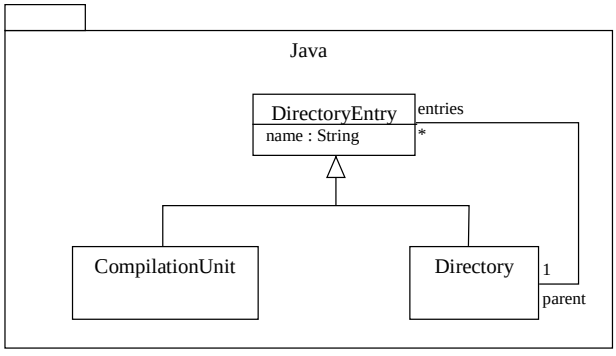

Figure 72 – The Java Model

The translator is required to map each node (TNode) of the tree to either a Directory or CompilationUnit. The tree nodes must indicate which component it is mapped to and the name of that component. We use the ‘data’ attribute of the TNode to carry this information by assigning a Pair of string values to it. The first string indicates the type of node, and the second indicates the name. The specific implementation of the Tree model that interfaces to the JTree component can interpret these strings to provide the appropriate icon and text for visualising the node.

The mapping specification for this translator is shown in Figure 73.

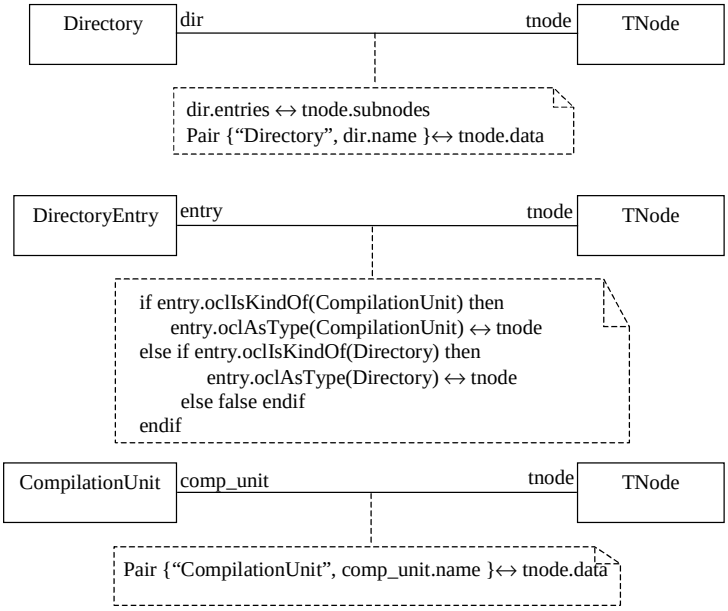

Figure 73 – Java ↔ Tree Mapping Specifications

This specification defines the required mappings. Each Directory and CompilationUnit are mapped to a TNode.

The Directory↔TNode mapping specifies that each entry in a directory is mapped to a subnode of the TNode to which it is mapped. It also specifies that the data of the TNode is mapped to a Pair indicating the type of node and the name of the Directory.

The DirectoryEntry↔TNode mapping forwards the mapping to either a Directory↔TNode or CompilationUnit↔TNode mapping.

The CompilationUnit↔TNode mapping specifies that the TNode data should be mapped to a Pair indicating the type of node and the name of the CompilationUnit.

Input

To generate an active implementation of this translator, we first provide an XMI document containing the specification (shown in Table 24).

```
<XMI version="1.1" xmlns:UML="org.omg/UML1.3">
  <XMI.header>
    <XMI.model xmi.name="Java_Tree" href="Java_Tree.xmi"/>
    <XMI.import xmi.name="Java" href="Java.xmi"/>
    <XMI.import xmi.name="Tree" href="Tree.xmi"/>
  </XMI.header>
  <XMI.content>
   <UML:Model name="Java_Tree"/>

    <UML:Association>
      <UML:Association.connection>
        <UML:AssociationEnd name="entry">
         <UML:AssociationEnd.type>
            <UML:Classifier name="DirectoryEntry" />
         </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="tnode">
         <UML:AssociationEnd.type>
            <UML:Classifier name="TNode" />
         </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
```

```
          <UML:Constraint.body xmi.value="true"/>
        </UML:Constraint>
      </UML:ModelElement.constraint>
    </UML:Association>

    <UML:Association>
      <UML:Association.connection>
        <UML:AssociationEnd name="dir">
         <UML:AssociationEnd.type>
            <UML:Classifier name="Directory" />
         </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="tnode">
         <UML:AssociationEnd.type>
            <UML:Classifier name="TNode" />
         </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
          <UML:Constraint.body
                xmi.value="    self.dir.entries->size
                             = self.tnode.subnodes->size
                           and
                               self.dir.name
                             = self.tnode.data.oclAsType(Pair).snd " />
        </UML:Constraint>
      </UML:ModelElement.constraint>
    </UML:Association>

    <UML:Association>
      <UML:Association.connection>
        <UML:AssociationEnd name="comp_unit">
         <UML:AssociationEnd.type>
            <UML:Classifier name="CompilationUnit" />
         </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="tnode">
         <UML:AssociationEnd.type>
            <UML:Classifier name="TNode" />
         </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
          <UML:Constraint.body
                xmi.value="    self.comp_unit.declarations->size
                             = self.tnode.subnodes->size
                           and
                               self.comp_unit.name
                             = self.tnode.data.oclAsType(Pair).snd"/>
        </UML:Constraint>
      </UML:ModelElement.constraint>
    </UML:Association>


   </UML:Model>
  </XMI.content>
</XMI>
```

Table 24 – XMI for Java↔Tree Mapping Specification

The OCL constraints are slightly different to those defined in the specification. This is primarily due to the use of the '↔' operator. The operator is shorthand for a longer OCL expression involving the use of the 'allInstances' operation and class templates (see Chapter 4). Considering that evaluating expressions that involve 'allInstances' is time consuming and that the Java language lacks a facility for defining class templates, an alternative to explicitly using the '↔' operator is required.

The use of the operator to map between Collections is altered to a comparison ('=') of the sizes of those collections. This will enable the constraint to provide appropriate events if objects are added to either collection and allow the evaluation of the

constraint to be ‘true’ if the collections are appropriately mapped. (Strictly, the constraint could evaluate to true when the collections have the correct number of objects but where those objects are not mapped to each other. However, the code for the translator does not allow that situation to occur, and hence the variation on the constraint is acceptable.)

The use of the operator to map between String values is altered to a straight comparison of the strings. This will enable the constraint to generate the required events and evaluate to true when the strings are correctly mapped.

The constraint on the DirectoryEntry↔TNode mapping is not included, as the generated mapping class is never instantiated. However, the specification is required, as other parts of the framework that are generated from it, are used.

Framework

The automatic framework generator produces a package ‘Java_Tree’ that contains two sub-packages named mappings and translator. The mappings package contains a MappingManager class and a class for each of the associations defined in the specification.

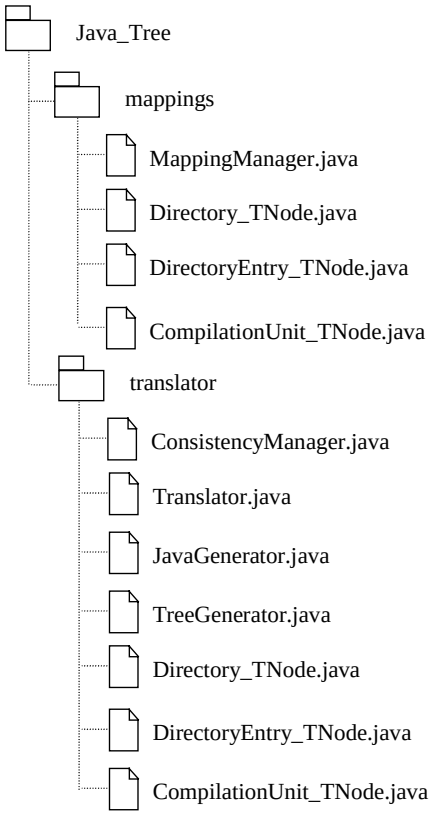

Figure 74 – Generated Files for Translator Implementation

The translator package contains a ConsistencyManager, a Translator, a Generator for each model, and a class for each mapping association. Figure 74 shows a tree view of these classes.

Manual Code

The parts of the implementation that must be subsequently hand coded are entered into the xxxGenerator classes and the classes for the mappings associations in the translator sub-package.

The constraints on the mapping association classes from the ‘mappings’ package perform inconsistency detection. The correlating classes in the translator package are for defining what actions should be performed as a result of detecting those inconsistencies.

The code for the Directory↔TNode mapping is shown in Table 25. (The light grey code is automatically generated, as is the case for subsequent tables of code presented in this chapter.) This illustrates the required manual additions to a consistency mapping class, for a simple mapping. Mapping constraint specifications of this simplicity could easily be generated automatically; however, as further examples show, it is not always as straightforward as this.

```
public class IDirectory_ITNode
  extends JavaASM_Tree.mappings.IDirectory_ITNode
{
  public IDirectory_ITNode(IDirectory dir, ITNode tnode, ITranslator trans) {
    super(dir, tnode);
    new EquiCollectionMonitor( constraint,
                               dir().entries(),
                               tnode().subnodes(),
                               trans );
    new EquiStringBufferMonitor( constraint,
                                 dir.name(),
                                 (IStringBuffer)((Pair)tnode().data()).snd(),
                                 trans );
  }
}
```

Table 25 – Code for Directory↔TNode ConsistencyMapping

This is a particular implementation of a consistency mapping between a Directory and a TNode. It extends the base mapping implementation (defined in the mappings package) and defines that actions that must be taken when the evaluation of the mapping constraint changes.

After calling the constructor for the base mapping class, an EquiCollectionMonitor is created. It is based on the mapping constraint and the two collections ‘dir.entries’ and ‘tnode_subnodes’, which are the two sides of the “↔” operator in the original specified constraint. The created object monitors a constraint for Add and Remove events whose source is one of two collections. When these events are received, it adds or removes the translation of the added or removed object to the other of the two collections.

Similarly, an EquiStringBufferMonitor is created to monitor changes in either the directory name or the second string in the tnodes data. The monitor detects event changes to either string and updates the other accordingly.

The code for the CompilationUnit↔TNode consistency mapping class is similar to the above, containing only the creation of an EquiStringBufferMonitor to map the name of the CompilationUnit to the data in the TNode.

The DirectoryEntry↔TNode consistency mapping is not instantiated by the framework, and hence no code is added to the class dropped by the framework.

In addition to the consistency mapping objects, the TreeGenerator and JavaGenerator classes require code to be added that defines how to create components from one side of the mapping, from the components on the other side. With each generator is the concept of a source model and a target model, the generator contains code that defines how to create components of the target model from components of the source model.

Table 62 shows the code for the JavaGenerator.

```
public class JavaGenerator
  extends AbstractGenerator
{
  IJavaBuilder _builder;

  private ConsistencyManager _mappings;
  public ConsistencyManager consistency_manager() { return _mappings; }

  public JavaGenerator(IJavaBuilder builder, IMappingManager mm) {
    _builder = builder;
    _mappings = (ConsistencyManager)mm;
  }

  public IDirectoryEntry createIDirectoryEntry( ITNode tn ) {
    Pair p = (Pair)tn.data();
    String s = ((IString)p.fst()).toString();
    if (s.equals("Directory")) return createIDirectory(tn);
    if (s.equals("CompilationUnit")) return createICompilationUnit(tn);
    throw new RuntimeException("Error:: Unknown TNode type - "+s);
  }

  public IDirectory createIDirectory( ITNode tn ) {
    IDirectory d = (IDirectory)consistency_manager().get2To1(tn.parent());
    if (d != null) {
      IDirectory subd = _builder.buildDirectory(d);
      subd.name().setTo((IString)((Pair)tn.data()).fst();
      consistency_manager().createMapping(subd,tn);
      return subd;
    }
    throw new RuntimeException("Error:: No mapping for parent of "+tn);
  }

  public ICompilationUnit createICompilationUnit( ITNode tn ) {
    IDirectory d = (IDirectory)consistency_manager().get2To1(tn.parent());
    if (d != null) {
      ICompilationUnit cu = _builder.buildCompilationUnit(d);
      cu.name().setTo((IString)((Pair)tn.data()).fst();
      consistency_manager().createMapping(cu,tn);
      return cu;
    }
    throw new RuntimeException("Error:: No mapping for parent of "+tn);
  }

}
```

Table 26 – Java Generator Class

A Generator consists of:

- a reference to a builder object, used for constructing elements of the target model;
- a reference to the translator consistency manager; and
- a number of createXXX methods, each of which should contain details on how to create a component for the target model given the corresponding component from the source model.

The body of the createXXX methods must currently be filled in manually. The code must define how to build a component of the target model in such a way that the mapping constraint between the two components is valid.

For this generator, the createIDirectoryEntry method defines how to create a DirectoryEntry from a TNode. The details of the creation are to create either a Directory or a CompilationUnit from the TNode, depending on the value of the first component of the Pair stored in the TNode's data attribute.

To create a Directory it is necessary to have a reference to another Directory that will form the parent of the new directory. This can be derived from the mapping of the

parent of the TNode from which we are creating the new directory (if there is no mapping for the TNode's parent, we cannot create the new Directory). The new Directory is built (using the builder and the retrieved parent directory), and then its name is set to the string provided in the second element of the TNode's data. Finally, a mapping is created between the source TNode and the new Directory.

To create a CompilationUnit, the code follows an almost identical sequence of actions: acquiring a parent directory for the new CompilationUnit; building the new CompilationUnit; setting its name; and creating a mapping between the source TNode and new CompilationUnit.

The TreeGenerator creates TNodes from DirectoryEntries, Directories and CompilationUnits. There are three 'create' methods, one for each source-model component type; the code can be found in Appendix E.

In conjunction with the rest of the automatically generated framework, these classes form an active implementation of a translator between the Java and Tree models. It is a straightforward mapping, but has been useful in illustrating the generated framework and illustrates the aspects that are necessary to fill in manually.

The code that is automatically generated for the Translator and Consistency Manager is included in Appendix E. The Consistency Manager contains a series of createMapping methods, which create and record the mappings between the pairs (or pairs of groups) of components from each model. It also contains flag used to indicate whether the translator is "switched on"; i.e. whether or not actions should be taken upon observation of events from the models. This is discussed further in the next example.

The translator contains a series of methods, each of which performs the translation of a component from one model into a component from the other. The translation is achieved by first looking to see if the source component already has a mapped value recorded in the consistency manager, if so then this value is returned. If no mapping exists, then the appropriate generator is used to construct a new (target) component from the source component.

The next subsection looks at another example that is more complex; it involves one-to-many mappings and the composition of two translators in a chain.

6.2.2 Visual State Machine Editor

This example illustrates the use of the translator specification and implementation technique for the provision of an editing environment for a visual language. The example is drawn from the authors use of the techniques to provide a graphical user interface for a stochastic automaton model checker. The model checker was built as part of an EPSRC project (reference number GR/L28890) and is documented in [Bryans_etal_00jan] and [Bryans_etal_00nov]; details regarding the resulting tool can be found in [Akehurst_etal_00].

Specification

The user interface requires an editor for specifying automata. An automaton consists of locations and transitions. Each state is named, and each transition may be labelled with a guard. Part of the abstract syntax model (ASM) for the automata in this project is shown in Figure 75.

To provide a user interface that enables automata to be designed it is necessary to define a concrete syntax and to define a mapping (translation) between the concrete components and the abstract ones.

The concrete syntax will use rectangles to represent locations and will contain a textual label to represent the name of the location. Transitions will be represented by lines with an associated label to represent the guard.

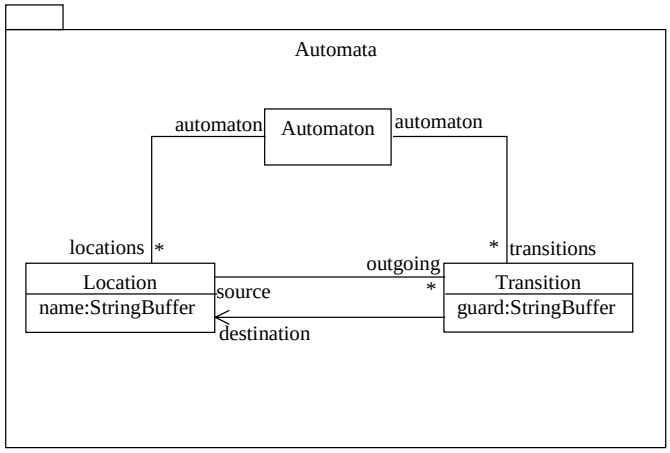

Figure 75 – An Automata Model

In accordance with the ideas presented in [Rekers_94] and [Bardohl_etal_99], the mapping between concrete syntax model and abstract syntax model for a visual language involves an intermediate Spatial Relationship Model (SRM). This model provides an abstraction from the specific shapes used in the notation, but retains the relationship between the symbols used in an expression (or diagram). The spatial relationship model for the automaton notation can be modelled using a labelled directed graph. The model of a labelled directed graph is a directed graph where each vertex and edge have an associated collection of labels. The UML representation of this model is shown in Figure 76.

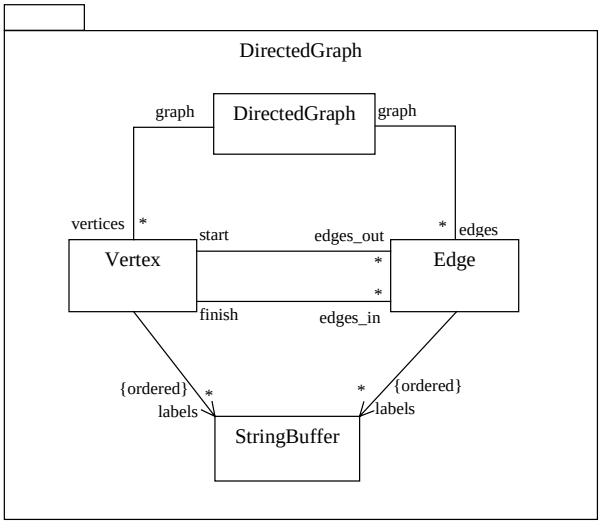

Figure 76 – A Labelled Directed Graph Model

To model the concrete syntax components we could use a specific implementation library such as the Java Swing library. However, a more platform independent option is to use a general model of graphical components such as the Standard Vector

Graphics model defined in [W3C_00aug]. A model for the SVG components used by this example is shown in Figure 76.

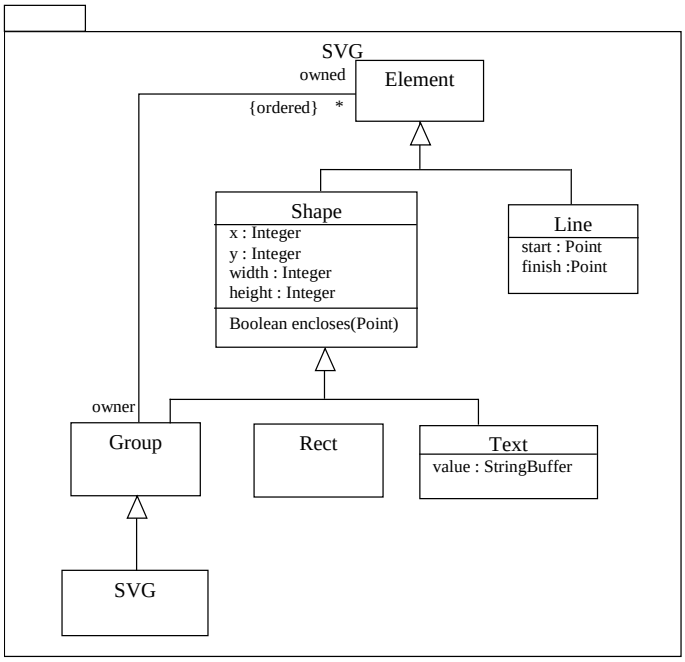

Figure 77 – A Partial SVG Model

The mapping between the abstract syntax Automata model and the spatial relationship graph model is shown in Figure 78. The mapping for this language is straightforward, consisting of three one-to-one mappings. Other more complex languages may require a more complex mapping between ASM and SRM.

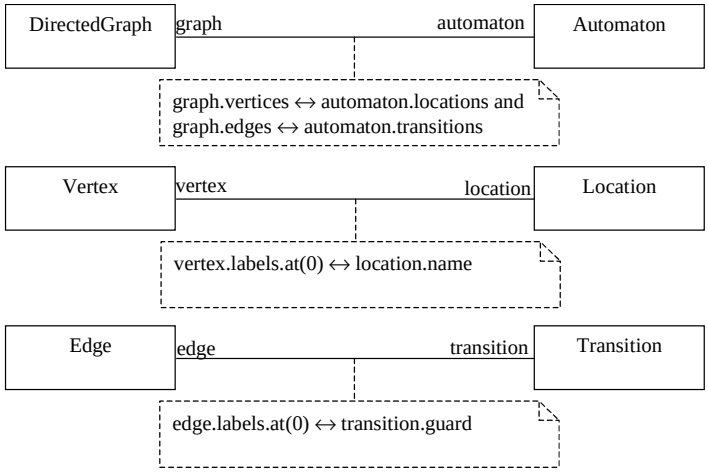

Figure 78 – DirectedGraph↔Automata Translator Specification

The mapping relates edges to transitions and vertices to locations. There is assumed to be only one label associated with each vertex or edge and this is mapped respectively, to the name or guard of the abstract component. The implementation of this translation is straightforward; hence, its discussion is not included, as it would not add to the explanation of this automatic implementation approach.

The mapping between SRM and concrete syntax model is more complex. Many different mappings could be specified; finding an efficient or ‘best’ one is the responsibility of the designer.

In this example, I have chosen to assume that each notation symbol is represented in SVG as a group (‘g’) element. There are two types of group, one consisting of a Rectangle and a Text element and one consisting of a Line and a Text element.

The enclosing ‘svg’ element is mapped to the DirectedGraph as specified in Figure 79. Each element in the SVG is assumed to be one of the two types of group (containing either line or rectangle). The set of elements is mapped to the union of the set of vertices and set of edges in the DirectedGraph; i.e. each group in the SVG is mapped to either a Vertex or an Edge from the DirectedGraph.

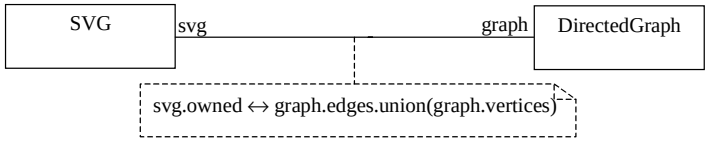

Figure 79 – SVG↔DirectedGraph Mapping Specification

Each Vertex is mapped to a Group and the elements within that group (Figure 80). The constraint on this mapping firstly ensures that the rectangle and text elements are members of the group; and secondly specifies that the value of the text element is mapped to the first label of the vertex.

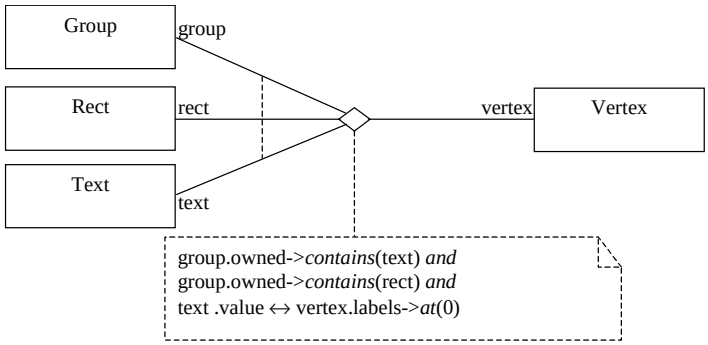

Figure 80 – (Group,Rect,Text) ↔ Vertex Mapping Specification

The mapping for an Edge is similar. Shown in Figure 81, an Edge is mapped to a Group and the Line and Text elements within that group. The value of the text element is mapped to the first label of the Edge.

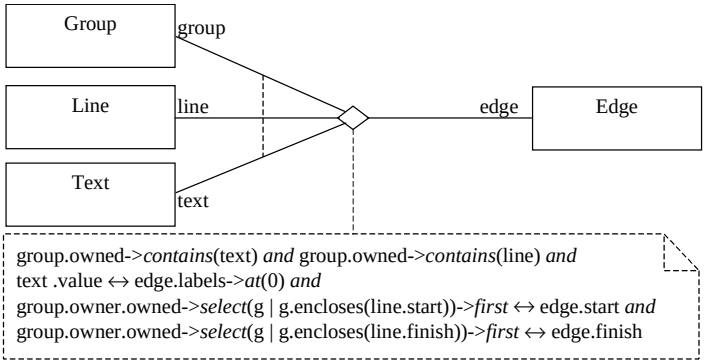

Figure 81 – (Group,Line,Text) ↔ Vertex Mapping Specification

The constraint for this mapping has two additional conjuncts; these define the connectivity of the expression. Lines that start at one Group and finish at another Group are interpreted to mean that the Vertices mapped to those Groups are connected by the Edge mapped to the Line. This is captured by the two parts of this constraint that select a Group whose dimensions enclose the starting or finishing

point of the Line involved in this mapping. The selected Groups must map to the Vertices at the start and finish of the Edge involved in this mapping.

In order that the above constraints can refer to a mapping between a Group and a Vertex we add the additional mapping shown in Figure 82. The constraint forwards the mapping to a mapping between the (Group, Rect, Text) tuple and a Vertex. A similar 'forwarding mapping' is required between a Group and an Edge.

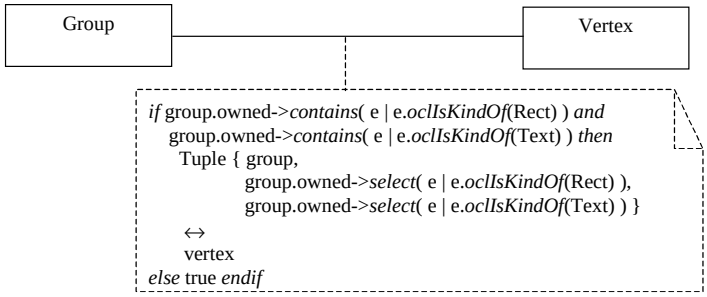

Figure 82 – Group↔Vertex Mapping Specification

The XMI files used as input to the translator generator can be found in Appendix F. There is one non-standard XMI element used – the 'Tie' element. This is used to encode the dashed lines that group AssociationEnds in an n-ary association (e.g. as used in Figure 81).

Framework

The generated framework of classes is similar for all translators. The only difference being the classes generated to encode the mapping associations and any additional tuple classes generated as a result of one-to-many or many-to-many mapping specifications (as discussed below).

The automatic generation of an active translator for this example involves a new aspect – mappings between multiple components. Both the (Group,Rect,Text)↔Vertex and (Group,Line,Text)↔Edge mappings are many-to-one mappings. The implementation framework requires mappings to be of the form one-to-one. To achieve this, the 'many' components are wrapped up in a tuple class to create a single object that can partake in the implementation of the translator framework.

As Java does not include facility for specifying typed Tuples via parameterised classes (templates), the automatic generator creates a specific class for each required tuple type. For this example, it generates two classes named tGroup$Rect$Text and tGroup$Line$Text as shown in Table 27.

```
public class tGroup$Rect$Text
{
  public tGroup$Rect$Text(IGroup group, IRect rect, IText text) {
    _group=group;
    _rect=rect;
    _text=text;
  }

  private IGroup _group;
  public IGroup group() {return _group;}

  private IRect _rect;
  public IRect rect() {return _rect;}

  private IText _text;
  public IText text() {return _text;}
```

```
  public int hashCode() { return   _group.hashCode()
                                 ^ _rect.hashCode()
                                 ^ _text.hashCode(); }
}
```

```
public class tGroup$Line$Text
{
  public tGroup$Line$Text(IGroup group, ILine line, IText text) {
    _group=group;
    _line=line;
    _text=text;
  }

  private IGroup _group;
  public IGroup group() {return _group;}

  private ILine _line;
  public ILine line() {return _line;}

  private IText _text;
  public IText text() {return _text;}

  public int hashCode() { return   _group.hashCode()
                                 ^ _line.hashCode()
                                 ^ _text.hashCode(); }

}
```

Table 27 – Two Tuple Classes

The generated tuple classes do nothing more than provide a wrapper for the contained objects. It is essential that any two tuple instances are considered equal if the objects contained in them are equal. This ensures that a tuple object created from its components can be correctly compared with other tuple objects created at a different time, from the same components. The default Java Object 'equals' performs a field by field comparison, which will ensure correct comparison; however, the 'hashCode' method must be overloaded to ensure duplicate tuple-objects are correctly retrieved from the collection classes used in the implementation.

Manual Code

Implementing the SVG↔DirectedGraph mapping requires a more complex approach than seen in the previous examples. The specification does not include a simple mapping between two collections. The mapping is between a collection and the union of two collections. Currently, the observable OCL library does not facilitate the observation of collections constructed as part of an expression[11], hence the components that make up the union must be separately observed. The code in Table 28 shows the implementation, which is discussed below.

```
public class ISVG_IDirectedGraph
  extends SVG_Graph.mappings.ISVG_IDirectedGraph
{
    ITranslator _trans=null;

    public ISVG_IDirectedGraph(ISVG svg, IDirectedGraph graph,
                                          ITranslator trans) {
       super(svg, graph);
       _trans = trans;
       Monitor m = new Monitor();
       m.watch(super.constraint,this,"observe");
    }

    public void observe(OclExpressionChangedEvent e) {
       IOclExpression expr = (IOclExpression)e.source();
       IObservableEvent oe = (IObservableEvent)e.original_event();
       if ( OCL.impl((IBoolean)expr.evaluate()) ) return;
       if ( ! _trans.consistency_manager().is_observing() ) return;
```

[11] Although it is believed that it may be possible to implement such functionality, but the task is left as future work.

```
       IMutableCollection col = (IMutableCollection)oe.source();
       if (col == graph().vertices() || col == graph().edges()) {
           if (oe instanceof AddEvent) {
              observe_graph_Add(((AddEvent)oe).new_value());
           } else if (oe instanceof RemoveEvent) {
              observe_graph_Remove(((RemoveEvent)oe).old_value());
           } else {
              System.out.println("Received Event: "+e);
              throw new RuntimeException(
                        "Error:: ISVG_IDirectedGraph received unknown event.");
           }
       } else if (col == svg().owned()) {
           if (oe instanceof AddEvent) {
              observe_svg_owned_Add(((AddEvent)oe).new_value());
           } else if (oe instanceof RemoveEvent) {
              observe_svg_owned_Remove(((RemoveEvent)oe).old_value());
           } else {
              System.out.println("Received Event: "+e);
              throw new RuntimeException(
                        "Error:: ISVG_IDirectedGraph received unknown event.");
           }
       }
    }

    void observe_graph_Add(Object added_obj) {
       Object new_obj = _trans.translate2To1(added_obj);
       if (new_obj instanceof tGroup$Rect$Text)
             svg().owned().add( ((tGroup$Rect$Text)new_obj).group() );
       else if (new_obj instanceof tGroup$Line$Text)
             svg().owned().add( ((tGroup$Line$Text)new_obj).group() );
       else if (new_obj instanceof IGroup)
             svg().owned().add( (IGroup)new_obj );
       else throw new RuntimeException(
             "Error:: unknown translation of "+added_obj+" as "+new_obj);
    }

    void observe_graph_Remove(Object removed_obj) {
       Object obj = _trans.translate2To1(removed_obj);
       IGroup g = null;
       if (obj instanceof IGroup) g=(IGroup)obj;
       if (obj instanceof tGroup$Rect$Text)g = ((tGroup$Rect$Text)obj).group();
       if (obj instanceof tGroup$Line$Text)g = ((tGroup$Line$Text)obj).group();
       svg().owned().remove(g);
    }

    void observe_svg_owned_Add(Object added_obj) {
       IOclAny new_obj = (IOclAny)_trans.translate1To2(added_obj);
       if (new_obj instanceof IVertex)
           graph().vertices().add(new_obj);
       else if (new_obj instanceof IEdge)
           graph().edges().add(new_obj);
       else
           throw new RuntimeException(
                      "Error:: ISVG_IDirectedGraph, unknown translation"
                      +" of "+added_obj+" to "+new_obj);
    }

    void observe_svg_owned_Remove(Object removed_obj) {
       IOclAny obj = (IOclAny)_trans.translate1To2(removed_obj);
       if (obj instanceof IVertex)
           graph().vertices().remove(obj);
       else if (obj instanceof IEdge)
           graph().edges().remove(obj);
       else
           throw new RuntimeException(
                      "Error:: ISVG_IDirectedGraph, unknown translation"
                      +" of "+removed_obj+" to "+obj);
    }
}
```

Table 28 – SVG↔DirectedGraph Consistency Mapping Class

The constructor of the mapping class creates a Monitor object that watches for events fired by the constraint expression and calls the method “observe” on ‘this’ object when events are received.

The “observe” method contains the actions to be performed as a result of the mapping constraint being inconsistent. The first two statements extract the original event that caused the expression to notice a possible change. Following these the expression is evaluated, if it evaluates to true then there is no need to perform any actions.

The next statement checks with the consistency manager to see if events should be acted on, if not, then no actions should be executed. This step stops event storms from being generated by mappings observing changes made by themselves as a result of actions performed in order to correct the inconsistencies. For example, in this case adding an object to the graph.vertices collection results in an action to add an element to the svg.owned collection, which is itself observed by this mapping. Without the check on the ‘is_observing’ flag, the mapping object would try to perform the actions for execution when elements are added to the svg.owned collection; which should only be executed if an external source adds to that collection.

Any actions to change mapped components models should be protected by this flag. The automatically generated framework sets the flag to false whenever a Generator ‘createXXX’ method is called (resetting it after the call). As the result of executing one of these create methods is assumed to produce correctly mapped components it is unnecessary for the mappings to act whilst additional model components are being generated. It is still necessary however, for the models to remain observable, as there may be other components (i.e. another translator) observing the generation of new components (see below, regarding the composition of the Automata↔DirectedGraph↔SVG chain of translators).

After determining that the consistency mapping object’s actions should be executed, the code determines the source of the observed event, which in this case is one of the three collections involved in the constraint – graph.vertices, graph.edges or svg.owned. Accordingly, a sub-method is called to perform the appropriate actions.

There are two possible event types generated by a collection – AddEvent and RemoveEvent – both of these events carry a reference to the object added or removed from the source collection. If the source of the events is either of the graph based collections then the translation of the added or removed object is added or removed to/from the svg.owned collection. Correspondingly, the translation of an object added or removed to/from the svg.owned collection is added or removed from either the graph.vertices or graph.edges collection.

The implementation of the tGroup$Rect$Text↔Vertex and tGroup$Line$Text↔Edge mappings simply require the value in the text element to be mapped to the first label in the list of labels for the Vertex or Edge. This is achieved using an EquiStringBufferMonitor as in the previous example.

The implementation of the SVG Generator classes is shown in Table 29.

```
public class svgGenerator
  extends AbstractGenerator
{
  IsvgBuilder _builder;

  private ConsistencyManager _mappings;
  public ConsistencyManager consistency_manager() { return _mappings; }
  public void setConsistencyManager(IConsistencyManager cm)
                                    {_mappings=(ConsistencyManager)cm;}

  public svgGenerator(IsvgBuilder builder, IConsistencyManager mm) {
    _builder = builder;
    _mappings = (ConsistencyManager)mm;
  }
```

```
  public ISVG createISVG(IDirectedGraph from) {
    ISVG to = _builder.buildSVG();
    consistency_manager().createMapping(to,from);
    return to;
  }

  public IGroup createIGroup(IVertex from) {
      tGroup$Rect$Text to = createtGroup$Rect$Text(from);
      return to.group();
  }

  ILayoutManager _layout=null;
  public ILayoutManager layout_manager() {return _layout;}
  public void setLayoutManager(ILayoutManager lm) {_layout = lm;}

  public tGroup$Rect$Text createtGroup$Rect$Text(IVertex from) {
    ISVG parent = (ISVG)consistency_manager().translator()
                                             .translate2To1(from.graph());
    IGroup group = _builder.buildGroup();
    group.setOwner(parent);
    layout_manager().layout(group,parent);
    IRect rect = _builder.buildRect();
    IText text = _builder.buildText();
    layout_manager().layout(rect,group);
    layout_manager().layout(text,group);
    group.owned().add(rect);
    group.owned().add(text);
    text.value().setTo( (IString)from.labels().at(OCL.Integer(0)) );
    consistency_manager().createMapping(group,rect,text,from);
    return new tGroup$Rect$Text(group,rect,text);
  }

  public tGroup$Line$Text createtGroup$Line$Text(IEdge from) {
      IGroup group = _builder.buildGroup();
      ILine line = _builder.buildLine();
            line.setStart( _builder.buildPoint() );
            line.setFinish( _builder.buildPoint() );
      IText text = _builder.buildText();
      group.owned().add(line);
      group.owned().add(text);

      IGroup start = (IGroup)consistency_manager().translator()
                                              .translate2To1(from.start());
      IGroup finish = (IGroup)consistency_manager().translator()
                                              .translate2To1(from.finish());

      line.start().setX( (IInteger)start.x().add(
                                  start.width().div(OCL.Integer(2))) );
      line.start().setY( (IInteger)start.y().add(
                                  start.height().div(OCL.Integer(2))) );
      line.finish().setX( (IInteger)finish.x().add(
                                   finish.width().div(OCL.Integer(2))) );
      line.finish().setY( (IInteger)finish.y().add(
                                   finish.height().div(OCL.Integer(2))) );

      text.value().setTo( (IString)from.labels().at(OCL.Integer(0)) );
      consistency_manager().createMapping(group, line, text,from);
      return new tGroup$Line$Text(group, line, text);
  }
}
```

Table 29 – SVG Generator Class

To create an SVG from a DirectedGraph, it is simply necessary to build the SVG and create the mapping. To create a Group from a Vertex, we must create the Group$Rect$Text tuple from the vertex, and return the group member of that tuple.

Creating a Group$Rect$Text tuple involves building each member of the tuple and putting the Rect and Text elements into the group. The text value must be set to that contained in the vertex label list and a mapping between the tuple and the vertex is then created.

The interesting aspect regarding this part of the translation is the need for additional information. There is no data stored within the DirectedGraph that indicates how to layout the graphical components, hence a LayoutManager is required. When a new graphical element is created, the layout manager is consulted to set the layout of that component within its parent. (The details of the layout algorithm are not relevant to this discussion.)

Similarly, to create a Group$Line$Text element from an Edge, the components are built and combined, the text value is set and the mapping object created. However, the layout of the line component is handled differently. The start and finish Points of the line are deduced from the positions of the elements mapped to each end of the Edge.

The DirectedGraph Generator performs the reverse generation, creating Vertices from Group$Rect$Text elements and Edges from Group$Line$Text elements. The code is very similar to that seen already and hence is not included here. The main interesting aspect is the generation of Edges. To create an Edge from the tuple including a Line, components must be detected at the start and finish Points of the Line. The Vertices to which they are mapped are used to set the start and finish of the Edge. Problems can occur if there is no component at the end of a Line or if there are multiple components at the end of the line. However, solving these issues is not relevant to the purpose of this example. One solution would be to indicate to the user that the addition of this line has caused an invalid situation (perhaps by altering the colour of the line!).

Translator Composition

The composition of the two translators in this system is made possible due to the observable qualities of the involved models. Each translator can be considered a separate system that responds to changes made to either of the translator's models.

The two translators are joined by a common model. Actions that alter the common model, executed by one translator, are detected as changes to the common model by the other translator, which in turn may cause other actions to be executed.

Without care, it can be seen that the two translators could set up an infinite loop, each reacting to the changes of the other. This problem can be avoided by making use of the 'is_observing' flag, contained in each Consistency Manager. The flag is set to false before a translator invokes any of the actions contained in a Generator and should be checked by any code that attempts to alter either model. Thus, when a translator is involved in performing updates, it effectively switches off its observation capabilities and consequently the live lock situation caused by the infinite loop described above cannot occur. The use of this flag also solves the problems of event storms as discussed in Chapter 5.

6.2.3 DirectedGraph↔Tree Translator

Much of the DirectedGraph↔Tree translator can be generated following the same process as for the previously described examples.

There is however, one complex issue not covered in the previous examples. This is in relation to the top-level mapping between DirectedGraph and Tree. This mapping involves the set of Vertices contained in the DirectedGraph, in order that the root of the tree can be mapped to an appropriate vertex in the graph.

The relationship between tree-root and vertex is set by selecting a particular vertex that has no incoming edges. Necessarily, this requires every vertex to be monitored, so that any change in the number of incoming edges of a vertex can be detected and the mapping of vertex to root be corrected.

The manually coded translator uses a particular implementation of a Collection that enables observation of every member of the collection, supporting addition and removal of members. This enables all members of the 'graph.vertices' collection to be observed. In addition, the directed graph model was extended so that each vertex fired events indicating that edges had been added or removed. Thus, changes to the number of incoming edges, of all vertices in the graph, could be monitored.

Currently, the Collection classes defined in the OCL library do not support observation of all their members; hence, this aspect of the example cannot be implemented. It is possible to extend the OCL Collection classes so that the required functionality is supported, but at this time, it is planned as future work.

6.3 Summary

This chapter has demonstrated a semi-automatic approach to implementing a translator directly from a UML/OCL translator specification. The implementation approach is illustrated using the Java programming language, although if the appropriate library support were to be ported, any object-oriented language would be useable.

The implementation is based on a framework involving an observable OCL library that is used to detect inconsistency in the mapping constraints. The actions to be performed, as a result of detecting an inconsistency, must currently be manually coded. For particular applications of translator, it may be possible to increase the amount of auto-generated code by analysing the constraints in more detail.

The translator generation scheme presented in this chapter is aimed at generating 'active' translators; it creates classes for each mapping relationship and a framework of classes that manage the mapping relationships. The constraint detection aspects of the mapping relationships are automatically generated using the observable OCL library and placeholders are created for entering the actions for responding to inconsistency.

Examples of the use of the auto-generation have been presented. These illustrate the mapping from translator specification to translator implementation. A potential item of future work would be to specify this mapping using the translator specification technique itself.

The first example (Java↔Tree) illustrates basic use of the automatic translator generator and shows the framework of Java classes that it generates. The second example (SVG↔Graph↔Automata) demonstrates use of the translator generator on a more complex example, which involves one to many mappings and the composition of two translators. This example also illustrates the use of the UML/OCL technique for specifying a Visual Language and shows how an editor for that language can be generated from the specification.

With reference to the examples illustrated, the XXXGenerator Classes are coded with the assumption that the models are built whilst the translator is operative. They could be enhanced by extending the generator code to construct the target model from an

already populated source model. This would require the XXXGenerator classes to be written in a top down manner and would require code that is more complex; however it would be a useful enhancement to the examples.

Each of the examples discussed in this chapter is included within an Appendix along with an example illustrating the use of the technique on a problem taken from the Permabase project (Appendix G).

The next chapter describes an evaluation of the translator implementation approaches along with an evaluation of the UML/OCL specification technique.

Chapter 7

Evaluation

The purpose of this chapter is to evaluate the proposed translator specification and implementation techniques with respect to the objectives they are suppose to meet. As set out in Chapter 1 the objectives of the thesis (paraphrased) are as follows:

1) To investigate whether UML can be used to specify model translations.

2) To investigate whether such specifications can be used to automatically generate active translator implementations.

The criteria by which the techniques are evaluated are chosen to test how successfully they meet these objectives. The specification technique is evaluated on its usability in the context of specifying translators between object-oriented models. The implementation technique is evaluated in the context of implementing a given specification and with respect to implementation related considerations.

The specification and implementation techniques are independently evaluated using a variety of different criteria that are set out in section 7.2. The section describes each of the criteria and discusses how they test techniques with respect to the objectives.

One general question to ask, specifically with respect to the specification technique, is "How easy is it to use?" This of course is a very subjective question and cannot be quantified. To overcome this difficulty, an evaluation framework of 'Cognitive Dimensions' is used to enable a discussion of the usability of the technique with reference to a set of standard terms. This framework is introduced in section 7.1 and is subsequently used as part of the evaluation.

Section 7.3 contains a brief overview and summary of the tasks to which the techniques have been applied. This outlines the experience of the author, regarding actual application of the techniques, upon which the evaluation results are based.

Section 7.4 contains the results of or the appropriate discussion surrounding the application of the evaluation criteria to the proposed specification and implementation techniques.

7.1 Cognitive Dimensions

Cognitive dimensions are a small set of 'non-specialist' terms that capture psychological aspects of languages and their use. These terms can subsequently be used as a framework in which to discuss the "usability" of a particular language, notation, or supporting tool.

The work started in 1989 by Thomas Green with his publication of an approach called "cognitive dimensions of notations" ([Green_89]). This has evolved into the cognitive

dimensions framework, a good description of which can be found in [Green_Petre_96].

The framework is 'task-specific', concentrating on the processes and activities a user of a language must perform, rather than on the finished product. Any cognitive language can be described using the dimensions (terms) giving a very high level description that indicates some major aspects of user activity.

This framework has been used for evaluation by companies such as Microsoft, Bently Systems and Synquiry Technologies and within a number of academic projects, such as those documented in [Burnett_etal_00], [Pane_Myers_96], [Britton_Jones_99], and [Shum_Hammond_94].

There are thirteen terms currently defined, each of which represents a concept that is orthogonal to each of the other terms. These thirteen terms are described in Table 30. The descriptions are taken or paraphrased from [Green_Petre_96].

Term	Description
Abstraction Gradient	What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
Closeness of mapping	What 'programming games' need to be learned?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Diffuseness	How many symbols or graphic entities are required to express a meaning?
Error-proneness	Does the design of the notation induce 'careless mistakes'?
Hard mental operations	Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what is happening?
Hidden dependencies	Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
Premature commitment	Does the user have to make decisions before they have the information they need?
Progressive evaluation	Can a partially-complete specification be examined to obtain feed back on "How am I doing"?
Role-expressiveness	Can the reader see how each component of a specification relates to the whole?
Secondary notation	Can the user make use of layout, colour, or other cues to convey extra meaning, beyond the 'official' semantics of the language?
Viscosity	How much effort is required to perform a single change?
Visibility	Is every part of the specification simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If

	the specification is dispersed, is it at least possible to know in what order to read it?

Table 30 – The Thirteen Cognitive Dimensions

The actual value of any *one* of these dimensions is related to the values of the other dimensions. For example, changing the Viscosity may require addition or removal of extra symbols, hence altering the Diffuseness.

It is not essential or necessarily possible to define a precise value for each of these dimensions, however they give a number of defined points related to usability that can be discussed in relation to any particular language.

7.2 Evaluation Criteria

The specification and implementation techniques must be evaluated against different criteria. This section describes the criteria that will be used for the evaluation of each technique.

7.2.1 For the Specification Technique

The objective relating to the translator specification technique is met by the definition of the specification technique described in Chapter 4 – A UML-based technique for specifying translators has been provided. The issues with respect to evaluation are therefore concerning the usability of this technique and of its technical expressiveness.

The ability to specify translations between object-oriented models is demonstrated by the use of the technique with examples. However, its effectiveness (i.e. how successful/usable it is for providing such specifications) is discussed within the framework of cognitive dimensions.

To evaluate the usability of the specification technique, it will be examined under each of the cognitive dimensions presented in section 7.1 above. This should give some indication as to how easy it is to specify a translator using the technique and how easy it would be for a user to learn the technique.

It is not possible to get an exact answer to the question of usability; it is a subjective problem. However, the discussion surrounding the cognitive dimensions will provide some indication of its usability.

To evaluate the technical expressiveness of the technique (i.e. can one use this technique to express a translator in any situation or are their limitations to its use?), a number of techniques could be adopted. Ideally, a formal analysis would be carried out to determine what could or couldn't be expressed using this technique, but that would be beyond the scope of this thesis. Alternatively, this specification technique could be compared, formally or informally, with other transformation specification techniques.

Such a comparison could be achieved by specifying a translation mapping between this technique and another; the alternative techniques discussed within this thesis for specifying translators are either Graph Grammars (GGs) or Triple Graph Grammars (TGGs). In order to specify a mapping from the UML/OCL technique into one of these, their meta-model and semantics must be used.

However, no meta-model has been formally defined for the Graph Transformation based techniques. Therefore, the approach used by this evaluation is an informal discussion comparing aspects of the UML/OCL technique with aspects of the GG and TGG techniques.

7.2.2 Implementations

Evaluation of the implementation technique can be slightly more objective. There are still subjective issues, such as "how easy is it to generate an implementation from the specification?" However, there are also attributes that can be precisely evaluated and compared, such as the size of the implementation or how memory hungry it is.

The objective relating to a translator implementation approach can be divided into five requirements, as follows:

1. The requirement for a translator implementation approach,
2. The requirement that the implementation is possible from a specification given using the defined translator specification technique,
3. The requirement that the implementation of the translator "actively" performs the translation – i.e. that the target model is updated whenever the source model is changed,
4. The requirement that the active translation occurs in both directions; i.e. that changes to either model cause the other to be updated.
5. The requirement that such implementations can be at least partially automated.

Requirements 1 and 2 are met by the content of Chapter 5. The chapter discusses two approaches to implementing a translator and each approach is discussed in the context of a UML/OCL specification of the translator.

The requirement for an 'active' translator implementation (3) is met by the Observer-based technique proposed in Chapter 5 and the dual way architecture of the mapping classes/objects provides the functionality of translating in both directions (requirement 4).

Chapter 6 demonstrates a semi-automatic approach to providing the observer-based implementation, thus meeting requirement 5.

The examples show the use of the implementation techniques. What is not as obvious is how successful or efficient the implementations are. That is the purpose of the evaluation criteria set out below.

7.2.3 Evaluation Criteria

The first of the criteria is the subjective issue of the **Ease of implementation.** This is the most subjective of the criteria; under this topic the evaluation will discuss how easy or how obvious it is to provide an implementation given a specification of the proposed form.

Objective criteria give quantifiable measures relating to the efficiency of the translator implementation technique. A number of characteristics of the translator and model specifications affect the objective measures of the translator performance. These are as follows:

- The size of the (source) model instance; i.e. the number of objects that form part of the particular model being translated.

- The size of the (source and destination) model specification; i.e. the number of classes that form part of the model specifications.
- The number of mapping specifications between the two models.
- The amount of inter-connectivity between the models occurring as a result of the mappings. This is related to the number of attributes involved in the constraints on each mapping; each attribute involved requires actions to be taken as a result of changes and this increases the inter-connectivity between models.

The evaluation will discuss the measures in relation to the translator implementation techniques and look at how the values would change with respect to these characteristics.

The objective measures that will be used are:

1. **Memory usage.** How much additional memory is used by invoking the translator and how is the amount of memory affected by the size of the model(s) being translated.
2. **Size of implementation.** How big is the implementation of the translator and how is the size affected by the size of the models? Measured by the number of lines of code required to implement it, this gives a measure of complexity, implying that the bigger the implementation, the more complex the implementation. This is not always a good indication of complexity, implementations can be simple and long, but it gives a quantifiable measure that can be used as part of a comparison.
3. **Speed of translation.** How long does it take for a model to be translated and how is the speed affected by the size of the model(s).

7.3 Use of the technique

The specification and implementation techniques have been used by the author to specify and implement a number of example translators. These are summarised in the following subsections. The observer-based implementation technique has also been used by D. Lewin as part of his MSc project ([Lewin_00]).

7.3.1 DirectedGraph ↔ Tree

A translator for converting between directed graph and tree structures has been specified and implemented as an example to illustrate the techniques throughout Chapters 4 and 5 of this thesis.

7.3.2 UML Actions ↔ RiscSim

One of the performance engines for the Permabase project is the RiscSim, Petri-Net based, engine. A translator from the Permabase central repository database (CMDS) into a RiscSim model was built as part of the project.

The CMDS included the specification of the behaviour of the modelled system. At the time of the project, the UML meta-model did not include suitable components for specifying the behaviour of the system. In order to store the required information a bespoke extension to the meta-model was developed specifically for use within the Permabase project.

During the time since the project terminated, the OMG has received various submissions in response to its RFP on UML Action Semantics ([OMG_98nov]). The eventual adoption by the OMG of one of these into the UML would be a preferable substitution for the extension used by Permabase.

A translator has been specified and implemented between part of one of the Action Semantics Submissions ([OMG_00aug]) and the RiscSim Performance engine ([Linington_99apr]). This illustrates the suitability of the techniques proposed by this thesis as a solution to the Permabase project requirements (see Appendix G).

7.3.3 UML Diagram Editors

Two different UML diagram editors have been implemented, making use of the proposed translator techniques. The diagrams are the UML Class and Sequence diagrams.

A diagram editor can be viewed as a two-stage translator, translating between concrete syntax and spatial relationship model and secondly between spatial relationship model and abstract syntax model (see Figure 83).

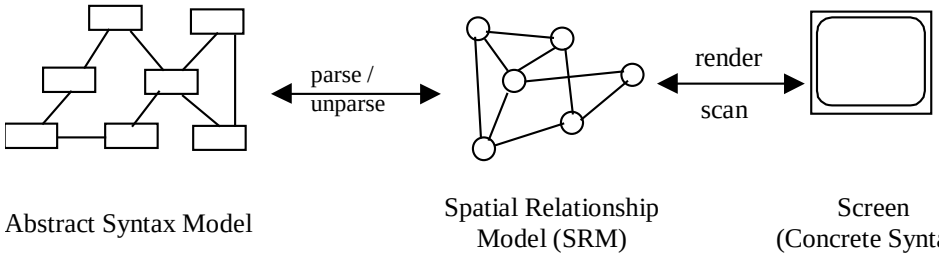

Figure 83 – Translator Architecture for a Visual Language

This two-stage translator architecture can be used to implement editors for many visual languages. The UML diagrams use parts of the UML meta-model as their abstract syntax models; the concrete syntax is formed from basic drawing components such as lines, arrows, boxes, or as defined by the components of standards like Precision Graphics Markup Language (PGML, [Adobe_98apr]) or Scalable Vector Graphics (SVG, [W3C_00aug]).

The SRM for simple class diagrams is a basic directed graph, as is the SRM for many other visual languages (such as State Machines, see below). However, to support the more complex parts of the class diagram notation (e.g. association classes) a more complex SRM is required that supports links between classes (vertices) and associations (edges).

The SRM for Sequence Diagrams is not a directed graph; however, a version of directed graphs with a small extension can be used. The vertical (life) lines of the objects forming the diagram behave in part like vertices and the message arrows as edges in a directed graph. However, the order in which the message-edges are connected to the object-vertices is of primary significance; it is the point of the diagram to show this order. Thus an extended from of graph model is required, which places an ordering on the collection of edges connected to the vertex.

7.3.4 Finite State Automata Based Model Checker

The translator based visual language architecture has been used to create a finite state automaton (variation of a state machine) editor as part of a project documented in

[Bryans_etal_00jan] and [Bryans_etal_99nov], investigating model-checking algorithms for such structures.

The editor forms part of a working tool that demonstrates the use of the model-checking algorithm. This translator has been used as an example to demonstrate the automatic implementation approach, in Chapter 6 (and see Appendix F).

7.3.5 Java Syntax Aware Editor

In addition to applying the approach to visual language editors, some investigation has been carried out into the use of the same style of architecture for textual languages. The concrete syntax and spatial relationship models for textual languages are respectively textual characters and an ordered sequence of tokens (groups of characters). The variation occurs in translating the token list into the abstract syntax model.

A small example has been implemented that maps a text document onto part of a Java abstract syntax model. The main issue involved with this example, is the specification of the mapping between a sequential list of tokens and a tree like abstract syntax model. This problem is general to text editors (implemented in this fashion) for many different languages.

7.3.6 Permabase Translators

As documented in Chapter 3, many translators were implemented as part of the Permabase project. These all used variations on the Visitor based translator implementation approach.

As stated, a specification technique was not used. However, the implementation of these translators provided the author with much of the background experience that provoked the requirements for an alternative approach.

Appendix G contains a specification of a segment of a translator that could form part of the Permabase tool. The specification defines part of the translation from a UML model of the system behaviour into a RiscSim (Petri-Net) based performance model (as discussed in secion 7.3.2).

7.4 Results

This subsection reports the results of evaluating the translator specification and implementation techniques according to the criteria outlined in section 7.2.

7.4.1 Evaluation of the Specification Technique

The following subsections discuss the proposed translator specification technique with respect to each of the cognitive dimensions and compare the technique with the GG and TGG alternatives.

Much of the usability of the translator specification technique is dependent on the usability of UML itself and hence the evaluation cannot avoid discussing the UML. However, an attempt is made to keep a distinction between the evaluation of the proposed specification technique and the associated discussion of the UML and OCL languages.

Some of the dimensions are more suited to the evaluation of a programming language or editor tool supporting a particular language. Where this is the case, it is indicated as such, and a limited discussion is included with respect to this evaluation.

7.4.1.1 Abstraction Gradient

An abstraction is a means of grouping one or more elements so that they can be treated as a single element. [Green_Petre_96] defines three classifications of abstraction gradient that can be applied to a language – abstraction-hating, abstraction-tolerant, or abstraction-hungry. These are based on the quantity of initial abstractions in the language and its readiness to desire or accept others.

The UML language falls into the classification of abstraction-tolerant. There are a number of initial abstraction mechanisms such as Package and Generalisation; however, these do not have to be used and a modeller using UML need not know about them in order to use the language. There are also higher order abstraction mechanisms in UML (e.g. stereotypes and profiles) that can be used to extend the modelling language itself possibly for creating other abstraction mechanisms.

The UML/OCL specification technique is also classified as abstraction-tolerant; it has one additional abstraction mechanism, the «mapped_to» operator added to OCL. This operator is used to refer to other mappings within the specification as a whole. Additionally there is a version of the operator used to relate two collections of model elements rather than forcing the specification of the relationship between every element in the collections (described in Chapter 4, subsection 4.3.8). Additional abstractions could also be added using the standard UML mechanisms.

The graph transformation techniques do not have abstraction mechanisms; there is no means to compose a transformation from simpler ones. A standard textual grammar can include in the RHS of a grammar rule a reference to another rule that represents an abstraction of a set of components. This issue is addressed in [Hoffmann_99] and the authors introduce a notion of transformation procedures that enable the composition of transformation rules.

7.4.1.2 Closeness of Mapping

This dimension is centred around the assertion that “the closer the language world is to the problem worlds, the easier the problem solving ought to be.” Ideally, things in the problem domain are mapped onto constructs in the language domain; it is generally accepted that textual languages are a long way from this, where as visual/graphical languages are surprisingly effective.

The proposed translator specification technique has both textual and visual elements in its use of the visual syntax of UML and the textual syntax of OCL.

The object-oriented approach of UML is presented as a modelling technique for representing ‘objects’ in the problem space and hence can be considered as having a high ‘closeness to mapping’. This is increased by the stereotyping feature of UML that enables the use of alternative icons that provide an even closer mapping to the problem domain.

OCL suffers from the same kinds of problems as any other mathematically based and textual language; its textual syntax does not provide any closeness to a problem domain. The designers of OCL, however, have tried to improve this by the use of

meaningful words instead of unusual textual symbols to represent the functions and operators; this does give it a higher rating than languages such as Z.

With respect to the translator specification technique, the problem domain is the requirement for a mapping between components from each model. The style of specification can be considered as close to the concept it is supposed to express. It makes use of an association (syntactically a connecting line) that links the related components, thus providing a clear visual indication that the two mapped components are related.

The use of OCL to elaborate on the details of the mapping is not a ‘close mapping’; it is not easy to see from the textual syntax what the constraints on the mapping are intended to be. Possibly the use of a more graphical constraint language (such as Constraint Diagrams [Gil_etal_99]) would improve this.

From the perspective of defining a mapping between models, the standard Graph Transformation approach cannot be considered as having a close mapping. There is no distinction made between the models, and no clear indication that such a specification contains a mapping between components from each model.

The TGG approach is an improvement, keeping the models separate, and the third correspondence grammar does convey an impression of relating two components. The use of a pair of numbers to specify the related components is a close mapping to the mathematical notation for a member of a relation, but it does not graphically show the connections between related components. Hence, TGGs still do not score very highly on the closeness to mapping scale, but are higher than standard Graph Transformations.

7.4.1.3 Consistency

This is concerned with the relationship between the different language concepts and constructs; i.e. if a person knows some of the language, how much of the rest can be guessed?

This evaluation will not delve into the issues of consistency within the UML and OCL languages or the inter-consistency between the two. Rather, it will look at how consistent the additional constructs, for specifying translators, are with respect to the existing constructs and their original use.

There are two aspects to the consistency of the translator specification technique, its consistency with UML and its consistency with OCL. From the UML perspective, the use of the stereotyped association is quite logical; although its additional semantics imply more than the standard association, it is still specifying a type of associative relationship between the mapped model components.

The additions to OCL are not, however, quite so consistent with the standard OCL syntax. The use of the «mapped_to» operator is inconsistent with the majority of the functions defined in OCL, which use a method call style of syntax :

<expression>.<function>(<expression>)

However, there are other binary operators (such as *and*, *or*, *implies*) that use the similar infix syntax of:

<expression> <operator> <expression>

The actual syntax of the «mapped_to» operator is also not entirely consistent with OCL syntax. The use of the guillemets (‘«’ and ‘»’) is inconsistent with other parts of

OCL and perhaps using the words without them (i.e. ‘mapped_to’ or ‘mappedto’) would be more consistent. However, the use of the guillemets is consistent with the declaration of a mapping in UML. The mapping is defined using a stereotyped association that is marked with the same (guillemeted) word/phrase and its use in an OCL statement enables it to be easily associated with the UML stereotype.

The alternative syntax (‘↔’) suggested for the operator is definitely not consistent with the OCL. The use of such a symbol is against one of the principles of OCL, which ensures that all OCL statements are expressible using standard ASCII characters.

The Graph Transformation approach to specifying a model translation is completely consistent; no additional notation or concepts are required. The TGG approach, on the other hand, is less consistent. The third (correspondence) grammar is interpreted differently to traditional grammars.

7.4.1.4 Diffuseness

The Diffuseness or Terseness of a language defines the number of symbols required to express a sentence. Different languages will be more or less terse depending on other factors of the languages, closeness of mapping for example. Visual languages inherently have a lower diffuseness as they make use of the positions of the symbols to carry extra information, hence requiring less symbols overall.

Very diffuse languages can be hard to understand due to being overwhelmed by symbols, on the other hand, a language which is too terse causes difficulties for a reader to distinguish between different sentences without very close examination.

This dimension is intended to measure the diffuseness of a language independently of these other factors. The number of symbols needed to express a problem solution is roughly quantifiable for any particular example. To fairly compare this value between different techniques, a standard defining what constitutes a symbol must be determined.

For this evaluation the graphical components, box and line are treated as basic symbols of the specifications and each word or other text symbol are considered as distinct symbols. The exact definition of what is counted as a symbol for the purpose of this evaluation is included in Appendix A.

Using this measure to evaluate the DirectedGraph ↔ Tree example, the values for each of the UML/OCL, Graph Transformation and TGG techniques are as shown in Table 31.

Technique	Total Symbols	Graphical Symbols	Textual Words/Symbols
UML/OCL (translator only)	127	25	102
UML/OCL (translator + models)	164	37	127
TGG (correspondence graph)	46	7	39
TGG (All three graphs)	147	35	112
Graph Transformation (transformation graphs)	176	71	105

Graph Transformation (transformation graphs + model graphs)	261	99	162

Table 31 – Diffuseness of DirectedGraph ↔ Tree Example

The Graph Transformation technique does not require the additional separate specification of the models; the models and translation are specified as a single grammar. However, statistics for the Graph Transformation technique are provided both inclusive of separate grammars for the models and exclusive of them.

By looking simply at the specification of the translator, the TGG approach clearly gives the most concise specification, with the UML/OCL and GT approaches being comparable.

However, if we include the specification of the models as well, each technique requires a similar number of symbols. The GT approach performs worst, as it duplicates the definition of the models; the extra model specifications are not strictly required in order to interpret the translator specification as they are with the other two techniques.

An interesting observation can be made by comparing the combination of model and translator specifications for the UML/OCL and TGG technique along with the translator only values for the GT technique (rows two, four and five of Table 31).

The values are very close, implying that the diffuseness of each technique is very similar.

The GT technique does use twice the number of graphical symbols, but the specification is split into two parts, one for each direction, hence requiring duplication (within the translator specifications as well as duplicating the specification of the models).

The OCL constraints vastly increase the number of symbols required for the UML/OCL technique. The textual nature of the language does not provide a concise specification and is the main contributor to the diffuseness of the UML/OCL technique.

7.4.1.5 Error-proneness

There is a difference between errors that are caused by making a ‘silly mistake’ or a slip and errors that are an incorrect specification, i.e. specifying something you didn’t mean to. This dimension is concerned with the ease of which ‘silly mistakes’ are made.

In general textual languages are very prone to this type of silly mistake; for example, correct pairing of braces, spelling mistakes, etc. Whereas visual languages tend to avoid these problems by, for example, using a single icon to represent a structure and not requiring the user to separately enter the end position (i.e. the closing brace). Mistakes in a visual language, such as drawing a line to the wrong box, are more easily detected than mistakes such as the pairing of braces.

Neither the UML, Graph Transformation, or TGG techniques are particularly prone to these mistakes. The OCL however, due to its textual nature does suffer from the problem; particularly with respect to pairing braces correctly in long constraints.

The UML/OCL translator specification technique often requires long constraints to specify the mappings and hence suffers from a high error-proneness rating.

7.4.1.6 Hard mental operations

It is very easy to write an ambiguous sentence that is grammatically and semantically correct, but is, nevertheless, very difficult to understand. The authors of [Green_Petre_96] use the following example that nicely indicates the problem:

> "Unless it is not the case that the lawn-mower is not in the shed, or if it *is* the case that the oil is not in the tool-box *and* the key is not on its hook, you will not need to cut the grass.…"

To understand what exactly is meant by this statement is a 'hard mental operation'; this is fine for word games, but is not desired in a useable design or programming language.

Two issue that contribute towards hard mental operations in a language:

- bad notations, i.e. is the notation a good way of expressing the concept; and
- combining two or three (badly notated) concepts, which vastly increases the difficulty of understanding.

A good test is to look at the combination of two or three constructs and ask how comprehensible such a sentence is; and secondly to determine if there is an alternative way of expressing that sentence (it may just be a hard idea to grasp).

This evaluation will not address the issues of whether or not the UML and OCL have adopted good or bad notations; that would be a controversial issue that is not the purpose of the evaluation. Rather it is interested in whether or not the translator specification technique, that uses these notations, involves hard mental operations.

The UML part of the notation is straightforward; the association clearly defines which components are involved in the mapping relationship. The OCL part however, is less obvious. An experienced OCL user would find the specifications easier to read, but may still experience some 'hard mental operations' with respect to the combination of multiple mapping relationships.

The mappings can be (effectively) nested by using the «mapped_to» operator, which states that a mapping relationship must exist between the components on each side. In such specifications it can be complex to remember what constraints have been specified on the referred to mapping and what must be additionally specified in this mapping. This gets worse with multiple levels of reference or when more than one mapping is referred to.

Another factor that affects whether or not a language involves hard mental operations is the past experience of the user and whether or not the user can adopt the correct 'mind set' for using the language. A designer who has been used to programming in object-oriented languages will probably find UML modelling fairly natural, whereas, a functional programmer is likely to find it harder.

This is not mentioned in [Green_Petre_96] as a factor to consider, but from the perspective of this thesis, it is particularly important. The objectives of the translator specification technique require it to be applicable to object-oriented models and are thus aimed at a community that is used to object-oriented design. Consequently, the UML/OCL solution that uses object-oriented concepts is likely to be easier for the user community to understand than the Graph Grammar based techniques.

7.4.1.7 Hidden dependencies

A hidden dependency is a relationship between two components that is not fully visible.

Within the UML/OCL translator specification technique there are hidden dependencies between the mapping specifications when one mapping refers to another as part of the constraint; the referred to mapping does not indicate ('know') that it is referred to.

Also, the mapping relationships are hidden from the model specifications. This is actually part of the requirement, that the model components do not have to be altered in order to form part of the mapping specification. However, it does mean that if they are changed, the mapping specification may no longer be valid and there is nothing to indicate to the user that the mapping must be updated when the model components are changed.

The same is true for the TGG approach, where the correspondence grammar may need updating as a result of changing the model grammars. The graph transformation approach does not have these dependencies hidden, in fact they are explicitly used as part of the mechanism for forming the translator specification. However, they are considered a drawback of the technique that clutters up the specification.

7.4.1.8 Premature commitment

This dimension is concerned with the amount of 'look-ahead' necessary whilst using the language. Typically, visual languages based on boxes and lines have less commitment to the creation order than text based languages, though some look-ahead problems can occur. [Green_Petre_96] identifies four forms of commitment as follows:

1. Commitment to layout – Does the user need to imagine (sketch on paper) the whole sentence before starting in order to achieve a reasonable layout?
2. Commitment to connections – A particular aspect of visual language layout is minimising cross over of connection lines.
3. Commitment to order of creation – Does the order in which constructs are created ('placed on the page') imply a particular semantics which is subsequently fixed in relation to the original creation order?
4. Commitment to choice of construct – Can a user change a construct into a different one after it has been created; i.e. changing a while loop into a repeat-until?

Both UML and the GG based techniques suffer from these to a certain degree, however good tool support for the language can minimise these problems.

Specifically with respect to translator specifications, the UML/OCL technique involves the 'mental operations' of looking ahead to determine which constraints to place on a certain mapping and which to place on sub-mappings that are referred to using the «mapped_to» operator.

7.4.1.9 Progressive evaluation

With respect to a programming language, progressive evaluation is the ability to execute a partially written program; this is often achieved by using procedure or function 'stubs'. Green and Petre assert that it is essential for novices to be able to

progressively evaluate their “sentences” and although experts can live without it, they find it useful.

This dimension is not applicable to the specification techniques in themselves; it must be applied to a tool that supports writing specifications in this notation. This dimension can subsequently refer to the ability to see the specification, as it is written (rather than the very old fashioned command-driven approach to producing graphical diagrams). This dimension can also be applied to a tool that supports evaluation of a specification, in which case one can ask whether or not a partial specification can be evaluated.

Partial evaluation of a UML/OCL translator specification depends on whether or not basic UML and OCL expressions can be evaluated. If evaluation of UML and OCL is possible, then it is quite possible, given two object models, to state whether the constraints forming part of a translator specification are invalidated or not.

7.4.1.10 Role-expressiveness

This dimension is intended to describe how easy it is to answer a question about a language expression such as “what is this bit for?” It addresses the issue of readability, how easy is it for someone, who did not write the expression, to understand what is meant by it. Within textual programming languages, it is generally accepted that this can be enhanced by the use of meaningful identifier names, well-structured modularity, and secondary notations.

Similar enhancements are equally application to visual languages and can be applied to UML specifications. The translator specification technique is as expressive as the UML and OCL languages that it uses.

7.4.1.11 Secondary notation

A secondary notation is one that can be used by the author of a language expression to add information that does not form part of the actual language semantics. The most common forms are the use of comments and indentation (layout for 2D languages) styles in programming languages.

UML and OCL both provide means for writing comments. They are particularly valuable with respect to the translator specification technique to aid the explanation of the constraints on the mappings.

The layout of the mapping specifications is useful for grouping the components of each model to aid the visual indication of two sides of the mapping.

The lack of standard notation for Graph Grammar based techniques makes them easily extendable with any additional notation that helps explain the meaning of the specifications.

7.4.1.12 Viscosity

Viscosity is the resistance to local change of the language, i.e. how easy is it to change something in an expression. This can of course be vastly aided by good tool support, for example treating the connection lines of a box and line language like rubber bands rather than having to explicitly move their ends. This can however increase the amount of premature commitment required, such tools normally require the two end of the rubber band to exist before the link is created.

Moving away from the characteristics of tool support, the viscosity is heavily dependent upon the type of change being made. If one of the components at the end of a mapping relationship is changed for an entirely different one, many knock-on changes to the constraints will likely be required including possibly a complete rewrite. However, in comparison, changing the name of an attribute may only require a simple textual change in the constraint.

Green and Petre illustrate an evaluation and comparison under this dimension by timing how long it takes to make a change to a particular example. This approach cannot be taken here, as a supporting tool does not exist.

Considering these issues, no conclusive evaluation regarding the viscosity of the UML/OCL translator specification technique can be easily carried out, independently of a tool that supports it. It may be possible to count the number of changes required when implemented by hand, for a particular example, and then calculate which could be ignored due to automation (e.g. global search and replace). However, some changes are heavily dependent on the complexity of tool support provided. For instance, the number of changes can be affected by the level of support for automatic layout, ranging across:

- none (the specification is drawn in a basic drawing editor);
- basic support for retaining connecting lines when nodes are moved;
- full scale layout of the whole specification.

7.4.1.13 Visibility

This last dimension addresses the issue of being able to ‘see’ a specification without significant amounts of cognitive work. In particular, can two or more parts of the specification be viewed at the same time, for comparison?

This dimension is also heavily affected by tool support; ideally, given a large enough page a specification in any of the discussed approaches could be viewed in its entirety.

Within limits that are more realistic, the ability to view multiple parts of a specification simultaneously can be aided by the ability to break up the specification in to small self contained parts. This would need to be supported by a tool for finding individual parts in a large specification.

The UML/OCL technique can be split into individual mapping relationships and both Grammar based approaches can be split into individual rules. Hence, the visibility rating of the techniques do not show significant differences.

7.4.1.14 Other Issues

It is important within the specification of a mapping constraint to define what is **not** a valid mapping as well as what is a valid mapping. This becomes more important when using the specification to provide an implementation, as it aids the determination of what components to generate in order to create a valid target component or set of components.

7.4.2 Evaluation of the Implementation Technique

The evaluation of the implementation technique starts with the discussions based on the subjective ease of implementation and is followed by a discussion on each of the objective measures.

7.4.2.1 Ease of Implementation

As with any other task, creating your first translator implementation is naturally harder than your second, or third, etc. However, it is possible to look at the implementation process and determine how much is a trivial filling out of automatically generated templates or of following a straight forward process; against how much, translator specific, thought processing is required on behalf of the implementor.

The implementation template for the visitor-based approach is essentially a visitor to the source model; the process requires the visit methods to be filled out with the target model generation code. The details of each visit method must be deduced from the nature of the particular constraints relating to the model component being visited.

The manual approach to generating the implementation of the observer-based approach is described in Chapter 5. It involves some analysis of the constraints for each mapping relationship that determines which attributes (if altered) will affect the mappings constraint. This information is used to create a template of ‘observe’ methods that must be filled in with actions to correct the mapping or indicate its invalidity.

The implementation process can be vastly eased by using the automatic generation tool and observable OCL library as described in Chapter 6. Using these tools and technique, the manual contribution to providing an active translator implementation is reduced to filling in placeholders for the consistency mapping actions and the code for creating mapped objects from each other in the XXXGenerator classes.

The process of converting constraints into either generation code for the visitor-based translator or actions for the observer-based implementation has not yet been analysed sufficiently to form a standard (or automatic) implementation algorithm (see sections on future work).

The implementation process for either technique is non-trivial. Although the templates aid the implementation process, without a standard algorithm or set of heuristics that define the process explicitly, the complexity of the process is approximately proportional to the complexity of the constraints involved in the mapping specification.

7.4.2.2 Translation Speed

A direct comparison of this measure between the two implementation techniques is not possible due to the inherent differences between the techniques. I.e. it is an objective of the Observer-based technique that the speed be reduced to as close to instantaneous as possible.

The visitor-based approach requires a whole model as input and will take an amount of time to translate it that is proportional to the size of the model.

The observer-based approach builds the target model whilst a source model is being constructed, the time to create a translation is therefore ‘zero’ time, as soon as the source model is built, so too is the target model. To be exact, the time to completion of the target model, is the time taken to execute the actions resulting from observing the last change to the source model.

The quantity of these actions will depend on the complexity of the constraint they are attempting to validate as a result of the change. It is therefore essential to keep the

complexity of the constraints to a minimum. This is primarily achieved by specifying mappings between as few components as possible – ideally between only one from each model.

Within a single threaded environment, the translation time will also be increased by creating 'chains' of translators that propagate an event across a number of models. This will require actions to be carried out that update each model affected. Use of multiple threads can reduce the 'apparent' time with respect to updating a single model, but introduces the problems of concurrency (discussed in Chapter 5).

7.4.2.3 Implementation Size

This can be measured in terms of lines of code and a comparison can be made. Table 32 shows the number of lines of code for each of the techniques, for a couple of different examples. More examples would be ideal, but at this point in time only these two translators have been implemented using all approaches.

Translator	Visitor	Observer	Auto-Generated Observer	
			Total	Manual Contribution
DirectedGraph↔Tree	~100	~300	~400	~100
UMLAction↔RiscSim	~100	~200	~350	~100

Table 32 – Number of Lines of Code to Implement the Translator

The figures in this table show the approximate number of lines of code required to implement two different translator examples using each of the implementation techniques. The values are calculated for the visitor-based approach by adding together the values for the translator in each direction. The observer-based values are calculated from the sum of the values for each mapping class. The values for the Auto-generated code are split into two parts, the total number of lines of code making up the example and the number of lines that were added manually[12].

If this small sample is representative of the approaches in general we can see that the Observer based technique requires an implementation that is 2 to 4 times the size of the visitor based technique. The automatic approach creates even more lines of code; however, the potion of that code that has to be manually written is much smaller, comparable with the visitor based implementation.

The automatic support enables the manual effort to be focused on to the difficult parts of the problem. The parts of the implementation that are simply repetition of a template, using different names, are automatically generated; and the constraint evaluation code is carried out by the supporting library.

7.4.2.4 Memory usage

Obviously, this is proportional to the size of the models being translated, but for any one particular model, the difference in memory usage between the two translator implementation approaches is of interest.

One significant difference is the duration for which memory is used. The Visitor-based approach needs only to use extra memory, above that used to store the models,

[12] The figures, for the auto-generated code, are based on generating an implementation with the assumption that the OCL library supports the required functionality, see Chapter 6.

when the translation process is executing. Secondly, the memory used by the translator itself is fixed in size; it does not vary with the size of model translated.

The observer-based implementation, requires additional memory, used by the translator objects, continuously, i.e. the memory for storing the mapping objects. The amount of this memory is proportional to the size of the models; mapping objects are required for each group of model objects involved in each of the mappings.

7.4.2.5 Amount of Support

One of the requirements of the implementation approaches was to necessitate minimal changes to the implementation of the source and target models in order to support the translator implementation.

In addition to the size of the translator implementation, it is important to consider the amount of additional code that it is required to add to the model implementations in order to support each of the implementation approaches.

The visitor-based approach requires each model component to be visitable. This requires five lines of code to be added to each model component class (as shown in Table 33). Plus the implementation of a model specific visitor interface, the size of which is proportional to the number of component classes in the model.

```
import library.IVisitable;
...
class X
  ...
  implements IVisitable
{
...
  public void accept(IVisitor visitor) {
    ((IModelVisitor)visitor).visit(this);
  }
...
}
```

Table 33 – Additional Model Code for the Visitor-Based Implementation

Each model class must implement the IVisitable interface by casting the passed visitor object into the appropriate model visitor object and calling the visit method on it for *this* particular class.

The observer-based approach requires the addition of one fixed line, plus an additional two lines for every observable attribute of the model object or a change to the implementation class of collection attributes. An example is shown in Table 34.

```
import library.observable.*;
...
class X
  extends ObservableSupport
  ...
{
...
  private Object _attribute1;
  public Object attribute1() {return _attribute1;}
  public void setAttribute(Object o) {
    Object old = _attribute1;
    _attribute1 = o;
    fire( new ChangeEvent(this, “attribute1”, old, o) );
  }
...
  private Collection _collection1 = Adapter.collection( new Vector() );
  public Collection collection() { return _collection; };
...
}
```

Table 34 – Additional Model Code for the Observer-Based Implementation

Each class must extend the ObservableSupport class that is provided by the supporting library (or implement the IObservable interface in a similar fashion). Subsequently any modifier (mutator) methods must fire an event indicating the change made to the attribute. Collection based attributes can be implemented using the adapter provided by the library, which forms an appropriate observable collection. The automated approach to generating the observer-based translators requires slightly different classes to be used, but essentially perform the same tasks and must be used in the same manner.

In addition to these changes to the model classes, the use of the additional library components should be considered, although they don't directly cause a significant number of changes to the model components.

It is quite feasible to automate the process of adding the additional code required by either of the implementation techniques, provided a standard implementation pattern is used for defining the basic model components. Such a generator has been implemented, which provides an appropriately observable implementation of a model from a definition in UML encoded as an XMI file.

7.5 Summary

This chapter has provided a discursive evaluation of the translator specification and implementation techniques proposed by the thesis.

7.5.1 Specification Technique

The specification technique has been evaluated with respect to a number of cognitive dimensions, each of which addresses a particular aspect of the usability of a language. The evaluation has used these dimensions to provide a framework for the evaluation discussing the proposed use of and extensions to UML and OCL in the context of the translator specifications. Where appropriate, a comparison has been made with the Graph Transformation and Triple Graph Grammar approaches.

The results of the discussion within this framework are summarised in Table 35.

Cognitive Dimension	Evaluation Summary
Abstraction Gradient	The UML/OCL technique is abstraction-tolerant; the mechanisms of UML and the «mapped_to» operator provide some abstraction mechanisms. The Graph Grammar based approaches do not make use of abstractions.
Closeness of mapping	The use of the «mapped_to» association stereotype within the UML/OCL technique provides a 'close mapping' to the concept it represents. The textual OCL constraints do not give a close mapping, although the two sided «mapped_to» operator does imply its two sided meaning. The GG approaches do not give a close mapping, though the TGG approach is an improvement over basic Graph

	Transformation rules.
Consistency	The «mapped_to» stereotype is consistent with UML. The use of the binary «mapped_to» operator is less consistent with OCL, although it does have a few similar operators. The GT and TGG approaches are consistent with other GG usage.
Diffuseness	The TGG approach gives the most concise specification, although the quantity of graphical symbols is comparable with the UML/OCL technique. The OCL constraints do not provide a concise specification and are the main cause of the diffuseness of the UML/OCL technique. The Graph Transformation approach is the least concise.
Error-proneness	The use of the textual OCL language causes the UML/OCL technique to be highly error-prone; it is easy to make ‘silly mistakes’ when writing (necessarily) long OCL constraints.
Hard mental operations	The UML part of the proposed approach is straightforward, however the OCL constraints and the abstractions using the «mapped_to» operator can cause some ‘hard mental operations’. Even so, the UML/OCL approach would be easier to use within an object-oriented domain than one of the GG based techniques.
Hidden dependencies	Both the TGG and UML/OCL approaches involve hidden dependencies, where as the GT approach does not. These hidden dependencies are intentionally included to de-couple the model definitions from the translator definitions and to reduce the ‘clutter’ within a specification.
Premature commitment	This primarily depends on the type of tool support for a language and is less relevant to this evaluation. However, a certain amount of ‘look-ahead’ is required with the use of the «mapped_to» operator.
Progressive evaluation	This is dependent on tool support for the languages.
Role-expressiveness	The expressiveness of the UML/OCL technique is dependent on the expressiveness of the languages themselves. Sensible use of class, role and attributes names can enhance it.
Secondary notation	Both UML and OCL provide means to ‘comment’ the expressions; this is particularly useful in aiding the understanding of some mappings. Layout is also helpful.

	The lack of standard notation for the GG based techniques means that effectively any 'secondary' notation can be added.
Viscosity	This is dependent on tool support.
Visibility	This is heavily affected by tool support, however theoretically the UML/OCL, TGG and GT techniques all have good visibility – multiple parts of the specifications can easily be juxtaposed for comparison.

Table 35 – Cognitive Dimension Evaluation Summary

7.5.2 Implementation Techniques

The implementation techniques each have different advantages. As with all implementations memory usage must be played off against speed of execution.

The visitor-based approach has a long execution time vs. low memory usage and the observer-based approach has vice versa, a high memory usage vs. minimal execution time.

Both of the implementation approaches involve a complex conversion of the constraint specifications into appropriate generation code. Reporting errors as part of the conversion process is slightly more complex using the observer-based approach, although a straightforward solution is presented.

The observer-based approach has a number of complex issues that must be considered as part of the implementation process. In particular the avoidance of live-lock and other knock on effects of changing one or other model as part of the actions performed by a mapping object.

In practice, both implementation methods are complementary. The Active version gives continuous valid translations of the models given a starting position of two models being valid translations of each other. However, it is often the case that a translation is started from one partially populated model and the other needs to be generated before the active translation can be invoked. Traversing the original model using the visitor base translator is an efficient means of doing this.

7.5.3 Future Work

It would be interesting to develop some complexity measures for translator specifications, possibly based on the quantity and degree of connectivity between mapped components, and relate these to the implementation techniques.

This could form a means to predict the potential resource usage (memory or execution time) of the approaches and aid a designer in specifying more efficient translator specifications.

Chapter 8

Conclusion

This chapter concludes the thesis by summarising the work presented, in section 8.1. This is followed by highlighting the achievements in terms of the objectives defined in the introduction (in section 8.2) and finally (in section 8.3) by discussing possible future research that continues from that presented in this thesis.

8.1 Thesis Summary

Chapter 1 discusses a variety of applications of model translation within software engineering. Additionally it introduces the Object Management Group's (OMG's) Model Driven Architecture (MDA) initiative; this is a high profile framework in which the translation of models forms an important component.

The most mature technique for specifying translations (at this time) is that of Graph Transformations; unfortunately these are not based on the object-oriented modelling formalism, and hence are not compatible with the OMG's Unified Modelling Language (UML).

This thesis has investigated and presented an object-oriented, UML-based technique for the specification and implementation of model translators. The technique does not claim to be any more expressive than the Graph Transformation approach, however, it does provide an approach that is inline with current common practice in the software engineering community and the MDA initiative.

This thesis also proposes future work that will enable translation of specifications between the Graph Transformation approach and the proposed UML/OCL technique. If successful, this will enable the background and experience of the Graph Transformation community to be brought into common usage within software engineering as a whole.

The specification technique makes use of the UML and OCL to give a declarative specification of the relationship between components of the two models involved in the translation. Binary and N-ary associations are used to specify the relationships, the details of which are enhanced by the addition of OCL constraints.

The use of UML as the specification language means that this technique is also compatible with the upcoming introduction of XSLT used for translating between models defined using the popular and standardised XML.

The UML/OCL specification technique has been evaluated within a framework of cognitive dimensions (proposed in [Green_Petre_96]), and is seen to be as equally useable for specifying translators as the Graph Transformation approach. The main disadvantages of the UML/OCL approach are introduced by the use of OCL for

specifying details within the translator mappings, although this could be improved by using an alternative notation such as Constraint Diagrams ([Gil_etal_99]).

The most significant advantage of the technique proposed in this thesis, is its use of the standardised and commonly used UML and OCL. This makes the technique readily adoptable by those already familiar with the largely accepted standard notation for object-oriented modelling.

Two implementation approaches are discussed as ways of realising a translation from UML/OCL translator specifications.

The first approach is based on the Visitor and Builder patterns and is the approach that was used to implement the translators forming part of the Permabase project prototype. Although this approach is straightforward to use, it has two significant problems with respect to its use in the interactive Permabase environment. The main advantages and disadvantages are listed below:

Advantages

1. Minimal additions to the source and target model components are needed.
2. The size of the model instances processed by the translator are not limited by the amount of memory of the hosting processor.

Disadvantages

1. Translators implemented in this way are single step monolithic processes. They require re-executing every time the source model changes in order that the target model is kept up to date.
2. The time taken to execute the translation increases proportionally to the size of the model instance being processed.

To address these two disadvantages, a second implementation approach has been developed and presented. This second approach makes use of the observer pattern to ‘actively’ and continuously monitor the source model for changes, upon the occurrence of which, the target model is altered to accurately reflect the translation.

This Observer based approach breaks up the translation behaviour into a number of mini and localised translators that map a small number of components from the source model onto their equivalent target model components. These mini translators are activated by the occurrence of changes to the source model components that they monitor.

The problems with the Visitor based approach are solved as follows:

1. The mini translators actively monitor (observe) changes to the source model and execute a localised translation upon every change. Hence the target model is continuously updated and a monolithic translation is not required.
2. The sizes of the source and target models don’t affect the translation time. Each mini translator is affected only by the components it is responsible for translating. Larger models simply mean a larger number of mini translators, not an increase in translation time.

Of course, nothing is free and the cost of removing the relationship between model size and translator execution time is the cost of storing the mini translators, which does increase as the model size increases.

A feature of the second translator implementation approach is that is can be implemented as a two-way translator. This enables changes in the source model to be

reflected in the target model and vice-versa, changes to the target model can be reflected in the source model.

This feature is achieved by extending the Observer pattern to that of a “Multi-Observer”. Each of the mini translators observes changes to the components of each model and appropriately changes the respective components of the other model.

This enhancement expands the applicable scope of this style of translator implementation. In particular multi-view environments can be built using the technique, enabling each of the views to be continuously consistent with one another.

The manual implementation process from a translator specification to active implementation was recognised as complex and time consuming. To aid the implementation a semi-automated approach has been developed.

The automatic generation consists of two main parts:

- The use of an observable OCL library; to encode OCL constraints over Java objects such that any change to the value of the constraint can be ‘observed’.
- The automatic generation of a framework of classes from an XMI encoding of a UML/OCL translator specification; this provides the basis for a translator implementation.

The classes within the generated framework perform the detection of inconsistencies between the related models, in accordance with the source UML/OCL translator specification. These classes are extended to convert the inconsistency detection into translation by manually adding actions that perform appropriate transformations on the models whenever an inconsistency is detected.

8.2 Achievements

The objectives laid out in Chapter 1 have been met by the content of this thesis as described below.

Objective 1 is met by the UML/OCL technique for specifying model translators described in Chapter 4. UML and OCL are both object-oriented specification methods and the specification technique enables the translation relationship to be defined between two models that have been specified using UML.

Objective 2 is met by the Observer based implementation approach described in Chapter 5 and 6. The discussion contained in these chapters demonstrates how to create a translator implementation from a UML/OCL specification. The implementation consists of multiple “mini translators” that are capable of monitoring both models for changes. The occurrence of changes to either model cause appropriate updates to the opposing model. This ensures that the models are continuously valid translations of each other.

8.3 Future work

There are a number of possible areas for continuing the research and work documented in this thesis. Some of these are discussed in the following subsections.

8.3.1 Specifying and re-implementing the Permabase prototype.

The main problem with the Permabase project was (ironically) the performance of the prototype toolkit. The time taken to execute the one-step implementation of the translation processes became a serious limitation when applied to large system designs.

The active translation approach documented in this thesis is a solution to this problem. If the prototype were to be re-implemented using the proposed approach it would significantly improve the performance and usability of the toolkit.

This would enable more extensive use of the Permabase tool, over a larger number of case studies; thus giving more confidence in its use as part of the system design lifecycle. In particular it would enable further validation of the tool throughout the design lifecycle of a case study, including validation of predicted results against actual measurement taken from a deployed system.

8.3.2 Visual languages

Visual languages can be seen as a translation from a concrete syntax model to an abstract syntax model; the UML/OCL specification technique can hence be used to specify visual languages. Additionally, the multi-observer based implementation approach is suitable for implementing visual language editors.

Some initial work has been carried out in this area and published in [Akehurst_00]. This work could be extended to include investigation into the specification and implementation of visual languages (and editors) that are not based on the directed graph style of spatial relationship model associated with box and line based diagrams.

A suitable case study for this investigation would be the Constraint Diagrams defined in [Gil_etal_99]. These diagrams are based on the concepts of contours and regions as shown in Figure 84.

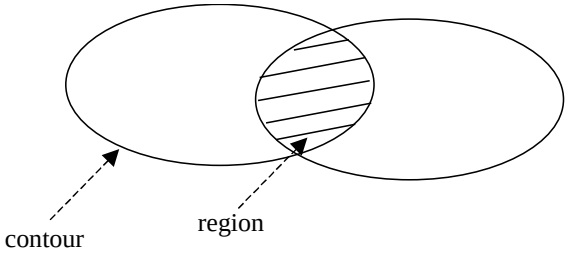

Figure 84 – Contours and Regions

Such diagrams cannot be mapped to a spatial relationship model based on directed graphs. Other work has been carried out by the authors of [Gil_etal_99] that identifies the basic concepts of the notation. The relationship of these concepts to the concrete visualisation and the abstract syntax components could be defined using the translator specification technique proposed in this thesis.

This approach to visual language specification could also be used to firm up the specification of the notations used to express diagrams in the UML. Currently the UML standard defines the notations using informal natural language. A formal definition could be defined by specifying a set of concrete symbols and their mapping to appropriate spatial relationship models and subsequently to the abstract syntax of the UML meta-model definition.

8.3.3 The formal relationship to Graph Grammars

Within Chapter 4, this thesis asserts that class diagrams, with the addition of OCL, are as expressive as a Graph Grammar. There is no formal backing for this assertion and work to produce evidence in support of it would be both interesting and provide a useful bridge between the Graph Grammar and object-oriented communities.

Continuing along this thread would be formal investigation into the respective expressive capabilities of Graph Transformation techniques and the UML/OCL technique for specifying model translations. Additionally, the specification of a translator between Graph Transformation specifications and UML/OCL translator specifications would aid this work and enable bodies of work in each area to be applied to the other.

In order to specify the translation it would be necessary to identify an abstract syntax model of graph grammar and graph transformation concepts. Such a model should ideally be one that is standardised by the graph grammar community.

Using this model a mapping could be specified between it and the UML meta-model, thus enabling conversion of graph grammar and transformation specifications into UML specifications.

Based on this mapping, tools could be built that provide both graph grammar based and class diagram based views on the same specifications. This would enable the two research communities to benefit from each others work. In particular it would bring the experience and techniques of the graph grammar community into the wide commercial community using the UML.

Appendix A

Scanning Rules

This appendix defines the symbol that are counted as distinct symbols when calculating the diffuseness of a model or translator specification as part of the evaluation recorded in Chapter 7. Symbols used for each of the UML/OCL, Graph Transformation or Triple Graph Grammar techniques, are included.

A.1 Textual Symbols

Each symbol shown in Table 36 is counted as a separate textual symbol.

Single Character Symbols	Composite Character Symbols	Character Sequence Symbols
.	«	Any number formed from a sequence of digits, [0-9]+
,	»	Any name (identifier) formed from a sequence of letters or underscore characters, [a-z,A-Z,_]+
:	==	
;	->	
(	::=	
)		
{		
}		
|		
λ		
*		

Table 36 – Distinct Textual Symbols

A.2 Graphical Symbols

Each symbol illustrated in Figure 85 is counted as a separate graphical symbol.

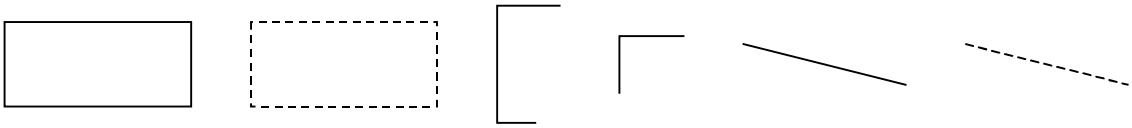

Figure 85 – Distinct Graphical Symbols

The symbols are any rectangular box drawn with a solid or dotted line; and any line or sequence of connected line segments drawn with dotted or solid lines.

Appendix B

UML Diagrams

The UML is essentially a collection of connected notations that are collectively used to describe the design of a software system. There are several different languages defined within the UML standard, which are referred to as 'Diagrams' or 'Diagram Types'. These diagram types are defined as follows:

- Use Case Diagram – illustrates the relationship between actors on the system and the use cases identified for that system. The diagram also shows extension and use relationships between the use cases.
- Static Structure (Class) Diagram – illustrates the static architecture (structure) of the system software. System components are generalised into 'classes' and the relationships between those classes are shown in a Static Structure Diagram. The classes can also be grouped into Packages; these diagrams can also be used to show the relationship between packages.
- Object Diagram – shows a particular network of objects, illustrating a group of objects, their attribute values and the links between the objects. This diagram can be considered a 'snapshot' in time of the state of a particular group of objects in the system.
- Interaction (Sequence and Collaboration) Diagram – shows the flow of messages (method calls) between a group of objects. The flow is shown either on top of an Object Diagram, which is subsequently referred to as a Collaboration Diagram, or using the alternative notation of a Sequence diagram. Both forms of Interaction Diagram illustrate the same semantic information, but use a different notation to do so.
- State Diagram (Statechart) – defines the states in which a class of object can be and shows the transition conditions that cause a change of state. The semantics of these state diagrams are based on Harel's Statecharts [Harel_87].
- Activity Diagram – is a special case of state diagram that exhibits primarily sequential flow. States are generally active states and transitions are generally triggered by completion of an activity in its source state.
- Component Diagram – used to show the software implementation components and inter-dependencies.
- Deployment Diagram – illustrates the run-time processing nodes, the node inter-connections and the objects and components that live (at run-time) on the nodes.

The two UML diagram types primarily used within this thesis are Class Diagrams and Object Diagrams. These are explained in more detail below.

B.1 Static Structure (Class) Diagrams

Static Structure Diagrams can show classes, packages, interfaces, objects and various relationships between them. If a static structure diagram shows only packages and their inter-relationships, it is sometimes referred to as a Package Diagram. Similarly, if only classes and their inter-relationships are shown the diagram is referred to as a Class Diagram. A static structure diagram that illustrates only objects gives an Object Diagram, although this is generally considered a separate diagram type and mixing objects and classes is rarely done.

B.2 Packages

A Package is a structuring component that is used to group other related components into a single unit. Packages can be nested inside one another and can be related by a dependency relationship. The dependency relationship indicates that one Package uses or depends on in some way one or more components from the other package.

The exact semantics of these UML components is not precisely specified in the standard, there are many ambiguities. Various bodies of research, such as [Schürr_Winter_98oct], attempt to rectify the situation giving a formal specification of the packages and the dependency relationship. However, such specifications have not yet been included in the standard UML definition.

The concrete syntax of a Package and Package Dependency is shown in Figure 86. The Package is represented by a rectangle with a ‘tab’ (smaller rectangle) on the top left corner. The name of the package is generally shown at the top centre of the main rectangle, although if the contents of the package are shown in the main rectangle, the name can be placed inside the tab.

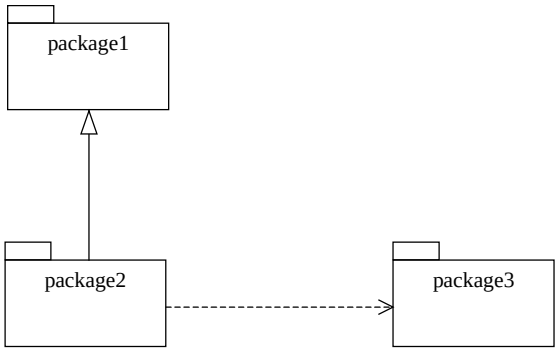

Figure 86 – Concrete Syntax of Packages and their Inter-Relationships

Package dependencies are shown as a dashed arrow with an open arrowhead pointing from the dependent Package to the Package upon which it depends. Any variation on the semantics of the dependency is shown as a stereotype label attached to the arrow (generally in the centre). A common variation is a generalisation relationship, indicating that the end Package is a generalisation of the one at the arrow start. Commonly this variation is shown using the solid line and empty triangle arrowhead also used to show generalisation between classes.

The contents of a package are usually shown enclosed in the main rectangle of the package or as a separate diagram. However, they can be shown in a tree like structure using branching lines with an encircled ‘+’ at the container end of the lines. Figure 87 shows the two single diagram methods for visualising a package and its contents.

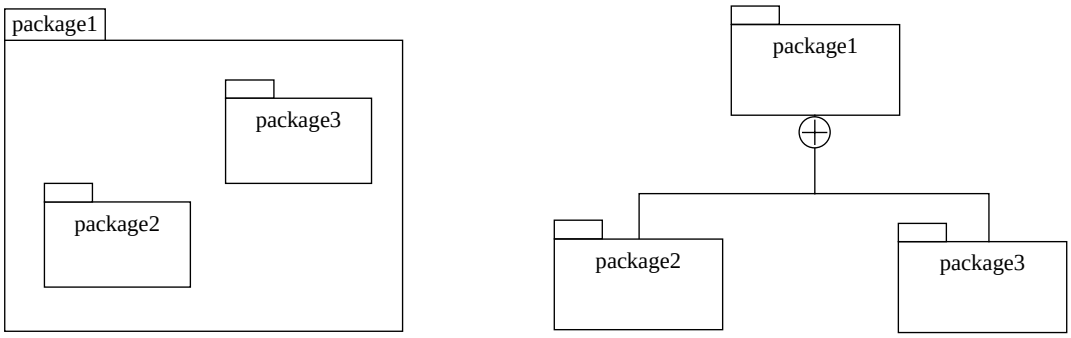

Figure 87 – Concrete Syntax for Illustrating Package Contents

B.3 Classes

A class is a representation of a concept within the system being modelled; it defines a pattern to which a set of objects conform, concerning the attributes, operations, behaviour and relationships of those objects. A class defines the attributes and operations for a particular set of objects, the behaviour is generally defined by a different view (or diagram) and the relationships between classes are show by a class diagram.

The concrete syntax for a class is shown in Figure 88. It consists of a rectangle divided into 3 compartments. The top compartment shows the name of the class and optionally any of the following: the stereotype of the class (if there is one); the package in which the class is defined; or tagged values. The other two compartments show the attributes and operations of the class.

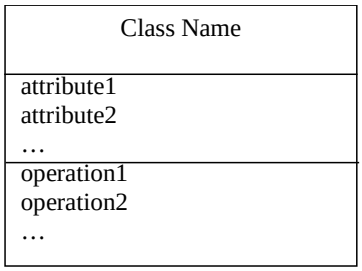

Figure 88 – Concrete Syntax for a Class

If the information regarding the attribute or operation specifications is unnecessary for a particular diagram, then the respective compartments need not be shown (as illustrated in Figure 89).

Two types of relationship can be defined between classes:

- Generalisations, which define a super-sub type (inheritance) relationship between two classes.
- Associations, which define an arbitrary relationship between any number of classes. The exact semantics of the association depend upon the adornments that can be added to the association in order to tighten the specification of the relationship.

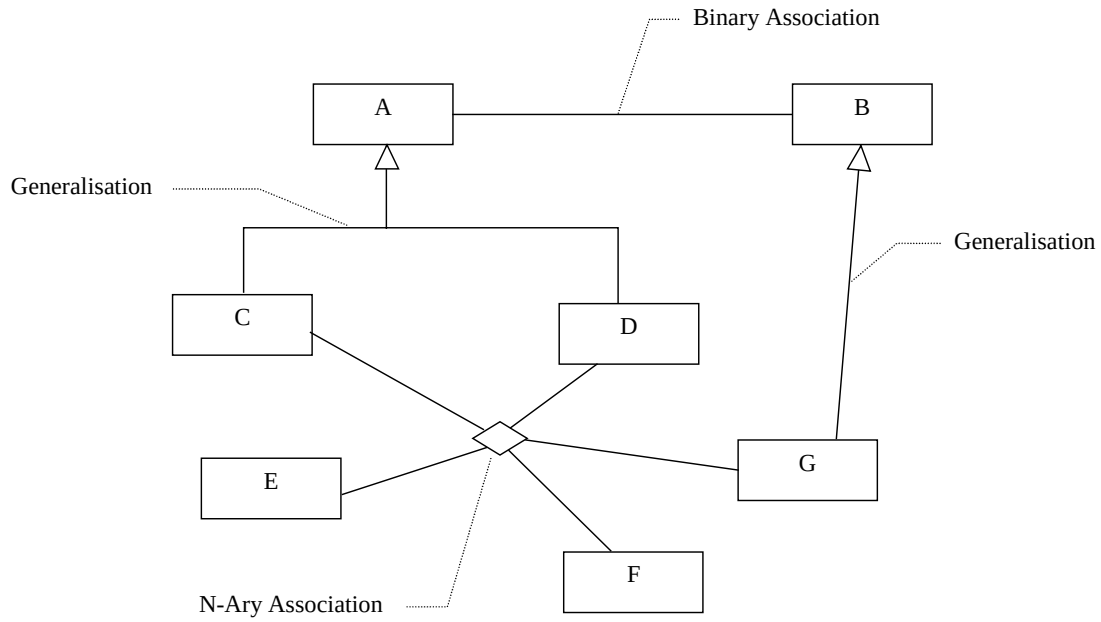

Figure 89 – Concrete Syntax's for Generalisation and Association Relationships

The concrete syntax for the generalisation and association relationships is shown in Figure 89. The Generalisation Relationship is illustrated using a solid line with an empty triangular arrowhead at the super class end of the line (as between classes G and B). If a tree like generalisation hierarchy is to be illustrated, branching lines can be used (as between classes A, C and D).

An Association between two classes (a Binary Association) is illustrated using a simple straight line between the two related classes. If the relationship is to associate more than two classes (an N-ary Association), an empty diamond is used as a central junction for lines linking each associated class (as between classes C, D, E, F and G).

B.3.1 Associations

The base form of an association doesn't say much about the relationship, other than that a connection between objects of the related classes can exist. Various adornments to the association add semantic meaning as follows:

- Rolename – any or all ends of the association can be given a name that provides a means to reference the object at that end of an instance of the association by other objects within the association.
- Cardinality – indicates the number of objects that must or may form part of an instance of the association. Further constraints can be added to an association end; those defined by the standard include the constraints 'ordered' and 'unique'.
- Navigability – puts a direction on the association that enables or disables visibility of components with respect to the others in the association.
- Aggregation – indicates that one object is an aggregate, the others in the association being the parts. The exact semantics of aggregation are widely disputed in the community, papers such as [Saksena_etal_98], [H-Sellers_Barbier_99] and [Civello_93] discuss the issues and define various interpretations.

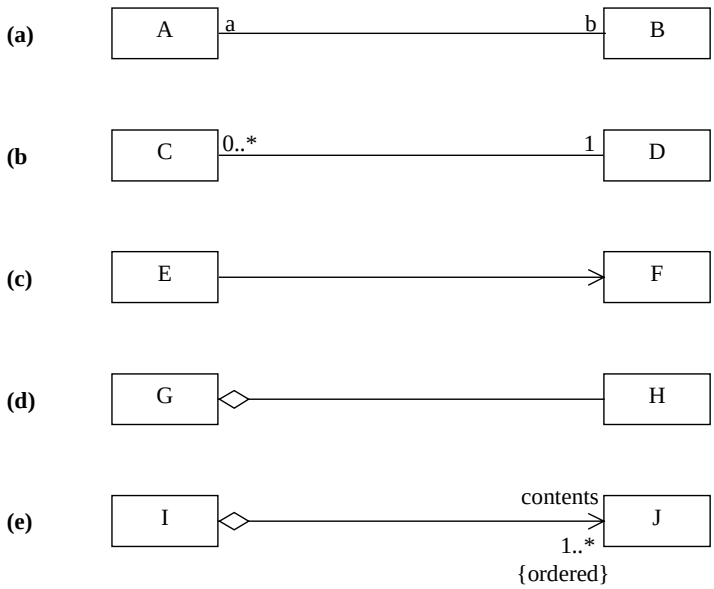

Figure 90 – Concrete Syntax for Various Adorned Associations

The concrete syntax for these adornments is illustrated in Figure 90. The first association (Figure 90a) shows rolenames (‘a’ and ‘b’) defined for each end of the association.

The next (Figure 90b) defines the cardinality (or multiplicity) of the classes at each end of the association. The cardinality of the C class states that zero or more C objects may be involved in the association. The cardinality of the D class states that exactly one D object must take part in the association.

The association in Figure 90c indicates that for any E and F objects linked as an instance of this association, the F object is visible to the E object, but not vice-versa. The E object is not navigable from the F object.

Figure 90d shows an aggregation association, it specifies that the G object is an aggregate including (an unspecified number of) H objects in its parts.

Finally, Figure 90e illustrates an association that combines all of these adornments. This association defines that the I object is an aggregate of one or more J objects; the collection of J objects is referred to within the I object by the name ‘contents’. This also illustrates the use of an additional constraint specifying that the collection is ordered.

When specifying an aggregation relationship, a black (solid) diamond can be used instead of a hollow one. This is generally considered to define a compositional aggregation, however the exact semantics of this form of aggregation is also widely disputed. In general (and within this thesis), the hollow diamond is used to indicate a reference relationship, where as the solid diamond indicates a compositional relationship carrying the concept of “by value” and ownership.

The compositional aggregation builds a tree like structure, each object can only be ‘owned’ by (be the ‘part’ end of) a single composite aggregation. Reference aggregations do not have such a restriction and can form any pattern of object inter-connections.

B.3.2 Interfaces

An interface is a specification of a set of operations that may be implemented by a variety of different classes. It provides a way of providing common access to a number of different implementations of the same behaviour.

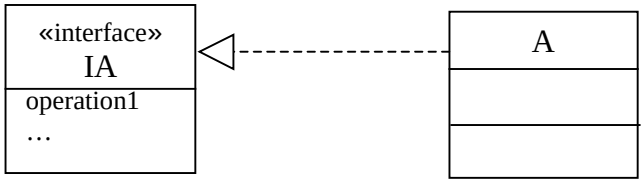

Figure 91 – Concrete Syntax for an Interface and Implementation Relationship

An interface is illustrated using similar syntax to a class, but with two compartments instead of one (Figure 91). The interface is a stereotype of a class and hence includes an indication as such in the top compartment along with the interface name.

A dashed arrow with a hollow triangular arrowhead is used to specify that a class implements a particular interface. Figure 91 shows that class A implements the interface IA.

B.3.3 Parameterised Classes

A parameterised Class (or Template) contains one or more unbound formal parameters; it defines a set of potential classes, each one specified by 'binding' the parameters to actual values. Figure 92a shows an example Parameterised Class and a corresponding Bound Class. Figure 92b shows the alternative shorthand that can be used to define a bound class, if a specific name for the class is not required.

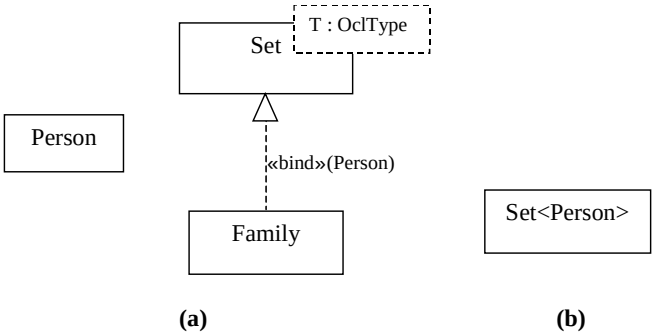

Figure 92 – Concrete Syntax for Parameterised and Bound Classes

A dashed Rectangle in the top right corner of a class is used to define the parameters of a Parameterised Class. The Bound Class can then be shown using an empty triangular arrowhead, dashed arrow with a «bind» stereotype indicating the values for the parameters; the arrow should point from the Bound Class to the Parameterised Class.

Alternatively, if a separately named class is unnecessary, a Bound Class can be defined by using the name of the Parameterised Class and the actual parameter values (enclose in '<>' characters) as the name of the Bound Class.

B.4 Object Diagrams

An Object Diagram shows a snapshot in time of the state of a group of objects and their interconnections. Two things form part of an Object Diagram, Objects and Links. Figure 93 illustrates an example Object Diagram.

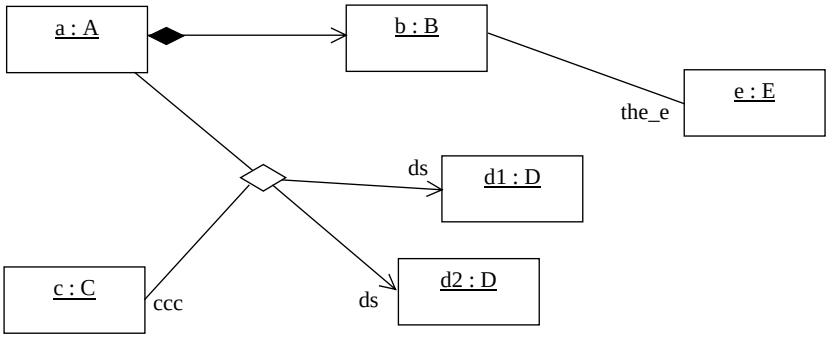

Figure 93 – An Example Object Diagram

The concrete syntax for an object is a rectangle with two compartments. The top compartment contains a name for the object and the name of the Class of the object; these are separated by a colon (‘:’) and are underlined. Either name may be omitted if unnecessary for a particular diagram. The bottom compartment contains values for the attributes of the object; this compartment may also be omitted (hidden) if its information is not required.

A link is an instance of an association; its syntax is a simple straight line, connecting the objects that form the associations instance. An n-ary link is shown using a diamond as the central point with lines to each participating object (in the same fashion as the n-ary association). Other than the cardinality adornment, any of the association adornments may be added to the line to increase the descriptiveness of the link being illustrated.

Appendix C

Graph Theory

C.1 Terms

The definitions of a graph from two different authors are as follows:

> "A graph G is an ordered pair of disjoint sets (V,E) such that E is a subset of the unordered pairs of V" [Bollobás_79], chapter 1.
>
> "a graph is a representation of a set of points and of the way they are joined up" [Wilson_72], chapter 1.

A graph is composed of two sets, the set of **Vertices** (or Nodes or Points) and the set of **Edges** (or Arcs or Lines) joining pairs of Vertices. An edge is said to **join** or connect two vertices. Vertices are **incident** with the edge connecting them. An edge that joins a vertex to itself is called a **loop**.

Two vertices are considered **adjacent** if there is an edge joining them. Two edges are **adjacent** if they share a common vertex.

The **degree** (or valence) of a vertex is a count of the number of ends of an edge that are connected to it. The **order** of a graph is the number of vertices, and the **size** of a graph is the number of edges. An **isolated** vertex has degree 0, and a **terminal** vertex (or endpoint) has degree 1.

A **walk** is an alternating sequence of vertices and edges, such that $edge_i$ joins $vertex_{i-1}$ and $vertex_i$. A **trail** is a walk in which each edge is distinct. A **path** is a trail in which each vertex is distinct. A walk has two **endvertices**, and a path can be considered a way of getting from one endvertix to the other. A **circuit** is a trail whose endvertices coincide. A **cycle** is a circuit with distinct vertices (different from path as there is an edge joining the endvertices).

A set of edges is **independent** if no two elements are adjacent, and a set of paths is **independent** if a vertex belonging to any two paths is an endvertex of both.

Two graphs are **isomorphic** if there is a one to one correspondence between their vertex sets, and such that for any pair of vertices from one graph, the vertices are joined by an edge if and only if they are joined by an edge in the other.

Two graphs are **homomorphic** if one can be mapped to the other by mapping multiple vertices from one graph onto a single vertex in the other.

C.2 Graph types

Based on constraints involving these terms, there are different variations of graph definition, each of which has different constraints and forms a different graph type.

A **simple graph** may only have one edge defined for each pair of vertices, and must not contain any loops.

A **connected graph** is a graph in which there is a path between every pair of vertices.

A **complete graph** is a simple graph in which every distinct pair of vertices is joined by an edge.

A **multigraph** can contain multiple edges connecting the same vertices, and can contain multiple loops.

A **directed graph** (or digraph) is one in which the edges are ordered pairs – they define a connection between vertices in a single direction.

A **bipartite graph** is one where the vertices can be divided into two disjoint groups and every edge joins a vertex from one group to a vertex of the other. The graph is a **complete bipartite graph** if every vertex from one group is joined to every vertex from the other.

An **acyclic graph** or **forest** is a graph with no cycles – a graph in which there is only one path between every pair of distinct vertices. A **tree** is a connected forest.

A **hypergraph** is a graph in which edges can join more than two vertices – the set of edges is a subset of the power set of vertices.

Appendix D

DirectedGraph-to-Tree Translator

This Appendix illustrates the complete manual active implementation of the DirectedGraph↔Tree translator.

D.1 DirectedGraph Model

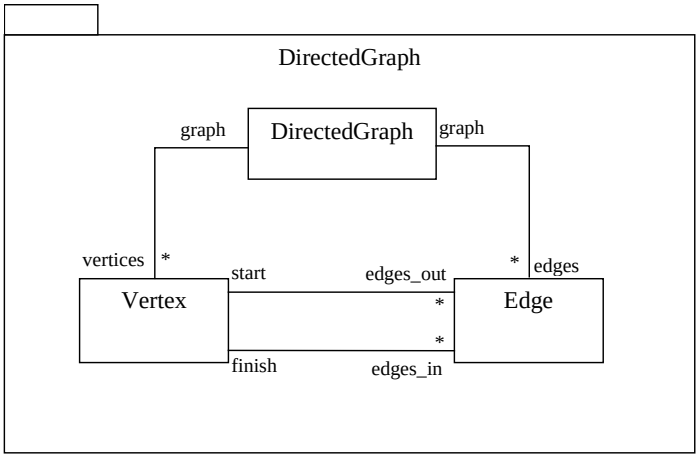

Figure 94 – DirectedGraph Package

D.2 Tree Model

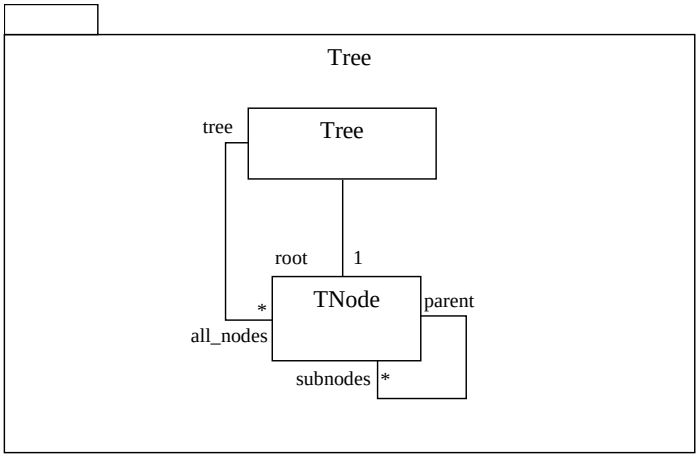

Figure 95 – Tree Package

D.3 DirectedGraph-to-Tree Translator Specification

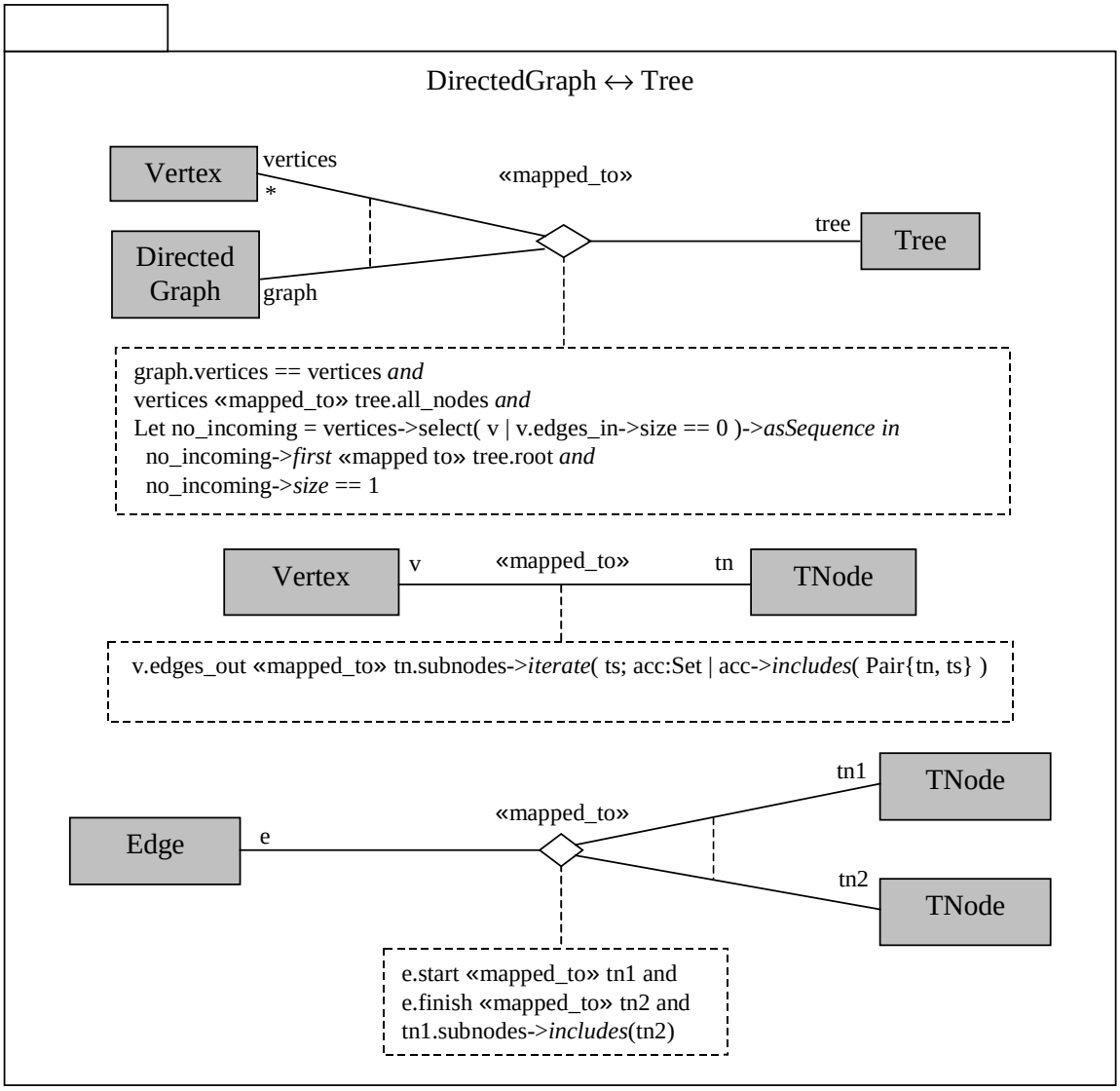

Figure 96 – DirectedGraph↔Tree Package

D.4 DirectedGraph-to-Tree Translator Implementation

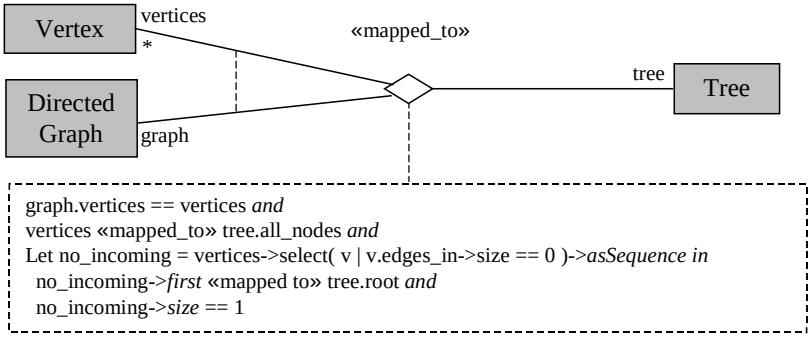

Figure 97 - DirectedGraph↔Tree mapping specification

D.4.1 Implementation Framework

```
public class DirectedGraph_Tree
  extends AbstractMapping
{

  private IDirectedGraph _graph;
  public IDirectedGraph graph() {return _graph;}
  private ITree _tree;
  public ITree tree() {return _tree;}

  //graph.vertices == vertices

  public Set vertices() {return graph().vertices();}

  public DirectedGraph_Tree(IDirectedGraph dg, ITree t) {
    _graph = dg;
    _tree = t;
    startObserving();
  }

  // Let no_incoming = vertices->select(v | v.edges_in->size == 0)
  private List no_incoming() { ... }

  public void observe_graph_vertices(AddEvent ev) { ... }

  public void observe_graph_vertices(RemoveEvent ev) { ... }

  public void observe_graph_edges(AddEvent ev) { ... }

  public void observe_graph_edges(RemoveEvent ev) { ... }

  public void observe_tree_root(ChangeEvent ev) { ... }

  public void observeAll_graph_vertices_edges_in(AddEvent ce) { ... }

  public void observeAll_graph_vertices_edges_in(RemoveEvent ce) { ... }

  public void observe_graph(IObservableEvent e) {
    if (e.name().equals("vertices")) {
      if (e instanceof AddEvent) observe_graph_vertices((AddEvent)e);
      if (e instanceof RemoveEvent) observe_graph_vertices((RemoveEvent)e);
    }
    if (e.name().equals("edges"))  {
      if (e instanceof AddEvent) observe_graph_edges((AddEvent)e);
      if (e instanceof RemoveEvent) observe_graph_edges((RemoveEvent)e);
    }
  }

  public void observe_tree(IObservableEvent e) {
    if (e.name().equals("root")) observe_tree_root((ChangeEvent)e);
  }

  public void observeAll_graph_vertices(IObservableEvent oe) {
    if (oe.name().equals("edges_in")) {
      if (oe instanceof AddEvent)
              observeAll_graph_vertices_edges_in((AddEvent)oe);
      if (oe instanceof RemoveEvent)
              observeAll_graph_vertices_edges_in((RemoveEvent)oe);
  }

  //--- AbstractMapping ---
  public Object object1() {return _graph;}
  public Object object2() {return _tree;}

  //---Observer---
  public void startObserving() {
    ((IObservable)graph()).addObserver(this);
    ((IObservable)tree()).addObserver(this);
```

```
    ((IobservableCollection)graph().edges()).addContentsObserver(this);
  }

  public void stopObserving() {
    ((IObservable)graph()).removeObserver(this);
    ((IObservable)tree()).removeObserver(this);
    ((IobservableCollection)graph().edges()).removeContentsObserver(this);
  }

  public void observe(IObservableEvent e) {
    if (e.source() == graph()) observe_graph(e);
    if (e.source() == tree()) observe_tree(e);
    if (graph().edges().contains(e.source())) observeAll_graph_edges(e);
  }

}
```

Table 37 – Implementation Framework for DirectedGraph↔Tree Mapping Class

D.4.2 Implementation of no_incoming

```
  // Let no_incoming = vertices->select(v | v.edges_in->size == 0)
  private List no_incoming() {
    List acc = new Vector();
    Iterator i = vertices().iterator();
    while (i.hasNext()) {
      IVertex v = (IVertex)i.next();
      if (v.edges_in().size() == 0) {
        acc.add(v);
      }
    }
    return acc;
  }
```

Table 38 – Actions to execute when an Edge is Added to a Vertex

D.4.3 Adding a Vertex

```
  public void observe_graph_vertices(AddEvent ev) {
    IVertex v = (IVertex)ev.new_value();

    // vertices <<mapped_to>> tree.all_nodes
    if ( manager().translate1(v) == null ) {
      ITNode tn = new TNode();

      ((Translator)manager()).createMapping(v,tn);
    }

    // no_incomin->first <<mapped_to>> tree.root
    List no_incoming = no_incoming();
    if ( no_incoming.size() > 0 ) {
      IVertex v1 = (IVertex) no_incomming.get(0);
      ((IValidateable)v1).setValid();
      ITNode tn = (ITNode)manager().translate1(v1);
      tree().setRoot(tn);
    }

    // no_incoming->size == 1
    if ( ( no_incoming.contains(v) ) && (v != no_incoming.get(0)) )
      ((IValidateable)v).setInvalid();
    }
  }
```

Table 39 – Actions to execute when a Vertex is Added to a DirectedGraph

D.4.4 Removing a Vertex

```
  public void observe_graph_vertices(RemoveEvent ev) {
    IVertex v = (IVertex)ev.old_value();

    // vertices <<mapped_to>> tree.all_nodes
    ITNode tn = (ITNode)manager().translate1(v);
    manager().removeMapping(v,tn);
    if (tn.parent() != null) {
      tn.parent().subnodes().remove(tn);
    }
```

```
    if (tn == tree().root()) {
      // no_incoming->first <<mapped_to>> t.root
      Iterator i = no_incoming().iterator();
      if (i.hasNext()) {
        IVertex vs = (IVertex)i.next();
        ITNode ts = (ITNode)manger().translate1(vs);
        tree().setRoot( ts );
      }
      // no_incoming->size == 1
      while(i.hasNext()) {
        IVertex vs = (IVertex)i.next();
        ((IValidateable)vs).setInvalid();
      }
    }
  }
```

Table 40 – Actions to execute when a Vertex is Removed from a DirectedGraph

D.4.5 Adding an Edge

```
  Public void observe_graph_edges(AddEvent ev) {
    IEdge e = (IEdge)ev.new_value();
    // vertices <<mapped_to>> tree.all_nodes
    // nothing
    // no_incoming->first <<mapped_to>> t.root
    // nothing
    // no_incoming->size == 1
    // nothing
  }
```

Table 41 – Actions to execute when an Edge is Added to a DirectedGraph

D.4.6 Removing an Edge

```
  public void observe_graph_edges(RemoveEvent ev) {
    // vertices <<mapped_to>> tree.all_nodes
    // nothing
    // no_incoming->first <<mapped_to>> t.root
    // nothing
    // no_incoming->size == 1
    //nothing
  }
```

Table 42 – Actions to execute when an Edge is Removed from a DirectedGraph

D.4.7 Adding an incoming Edge to a Vertex

```
  public void observeAll_graph_vertices_edges_in(AddEvent ae) {
    IVertex vertex = (IVertex)ae.source();
    // vertices <<mapped_to>> tree.all_nodes
    // nothing
    // no_incoming->first <<mapped_to>> t.root
    ITNode tn = (ITNode)manager().translate1(vertex);
    if (tn == tree().root()) {
      IVertex v = vertex;
      Set incomming = v.edges_in();
      while(!incomming.isEmpty()) {
        IEdge e2 = (IEdge)incomming.iterator().next();
        v = e2.start();
        incomming = v.edges_in();
      }
      ITNode new_root = (ITNode)manager().translate1(v);
      tree().setRoot(new_root);
      ((IValidateable)v).setValid();
    }
    // no_incoming->size == 1
    //nothing
}
```

Table 43 – Actions to execute when an Edge is Added to a Vertex

D.4.8 Removing an incoming Edge from a Vertex

```
public void observeAll_graph_vertices_edges_in(RemoveEvent re) {
  IVertex vertex = (IVertex)re.source();

  // vertices <<mapped_to>> tree.all_nodes
  // nothing

  // no_incoming->first <<mapped_to>> t.root
  // nothing

  // no_incoming->size == 1
  if (vertex.edges_in().size() == 0) {
    ((IValidateable)vertex).setInvalid();
  }
}
```

Table 44 – Actions to execute when an Edge is Removed from a Vertex

D.4.9 Changing the root TNode of a Tree

```
public void observe_tree_root(ChangeEvent ev) {
  ITNode tn = (ITNode)ev.new_value();
  IVertex v = (IVertex)manager().translate2(tn);
  ITNode tn_old = (ITNode)ev.old_value();
  IVertex v_old = (IVertex)manager().translate2(tn_old);

  // vertices <<mapped_to>> tree.all_nodes
  //nothing

  // no_incoming->first <<mapped_to>> t.root
  if (v == null) {
    v = new Vertex();
    ((Translator)manager()).createMapping(v,tn);
    graph().vertices().add(v);
  }

  // no_incoming->size == 1
  if (v_old != null) {
    if (v_old.edges_in().size()==0) {
      if (v_old != v) {
        ((Ivalidateable)v_old).setInvalid();
      }
    }
  }
}
```

Table 45 – Actions to execute when the root attribute of a Tree is Changed

D.5 Vertex↔TNode Implementation

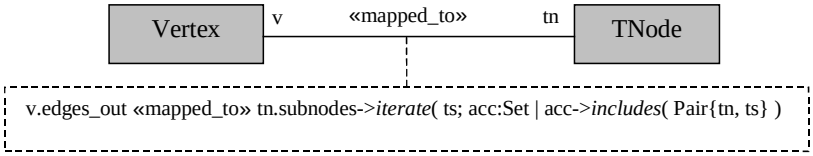

Figure 98 – Vertex↔TNode mapping specification

D.5.1 Implementation Framework

```
public class Vertex_TNode
  extends AbstractMapping
{
  private IVertex _vertex;
  public IVertex vertex() {return _vertex;}
  private ITNode _tnode;
  public ITNode tnode() {return _tnode;}

  public Vertex_TNode(IVertex v, ITNode tn) {
    _vertex = v;
    _tnode = tn;
    startObserving();
```

```
  }
  public void observe_vertex_edges_out(AddEvent ev) { ... }
  public void observe_vertex_edges_out(RemoveEvent ev) { ... }
  public void observe_vertex_edges_in(AddEvent ev) { ... }
  public void observe_vertex_edges_in(RemoveEvent ev) { ... }
  public void observe_tnode_subnodes(AddEvent ev) { ... }
  public void observe_tnode_subnodes(RemoveEvent ev) { ... }
  public void observe_vertex(IObservableEvent e) {
    if (e.name().equals("edges_out")) {
      if (e instanceof AddEvent) observe_vertex_edges_out((AddEvent)e);
      if (e instanceof RemoveEvent) observe_vertex_edges_out((RemoveEvent)e);
    }
  }
  public void observe_tnode(IObservableEvent e) {
    if (e.name().equals("subnodes")) {
      if (e instanceof AddEvent) observe_tnode_subnodes((AddEvent)e);
      if (e instanceof RemoveEvent) observe_tnode_subnodes((RemoveEvent)e);
    }
  }
  //--- AbstractMapping ---
  public Object object1() { return vertex(); }
  public Object object2() { return tnode(); }
  //---Observer---
  public void startObserving() {
    ((IObservable)vertex()).addObserver(this);
    ((IObservable)tnode()).addObserver(this);
  }
  public void stopObserving() {
    ((IObservable)vertex()).removeObserver(this);
    ((IObservable)tnode()).removeObserver(this);
  }
  public void observe(IObservableEvent e) {
    if (e.source() == vertex()) observe_vertex(e);
    if (e.source() == tnode()) observe_tnode(e);
  }
}
```

Table 46 – Implementation Framework for Vertex↔TNode Mapping Class

D.5.2 Adding an Outgoing Edge

```
  public void observe_vertex_edges_out(AddEvent ev) {
    IEdge new_edge = (IEdge)ev.new_value();
    //v.out_edges «mapped_to» tn.subnodes->iterate(ts,acc:Set|acc->includes(Pair{tn,ts}))
    Pair p = (Pair)manager().translate1(new_edge);
    if ( p == null) {
      ITNode tn = (ITNode)manager().translate1(new_edge.start());
      ITNode ts = (ITNode)manager().translate1(new_edge.finish());
      tn.subnodes().add(ts);
      ((Translator)manager()).createMapping(new_edge,new Pair(tn,ts));
    }
  }
```

Table 47 – Actions to execute when an outgoing Edge is Added

D.5.3 Removing an Outgoing Edge

```
  public void observe_vertex_edges_out(RemoveEvent ev) {
    IEdge old_edge = (IEdge)ev.old_value();
    //v.out_edges «mapped_to» tn.subnodes->iterate(ts,acc:Set|acc->includes(Pair{tn,ts}))
    Pair p = (Pair)manager().translate1(old_edge);
    manager().removeMapping(p,old_edge);
    ITNode tn = (ITNode)p.fst();
    ITNode ts = (ITNode)p.snd();
    tn.subnodes().remove(ts);
  }
```

Table 48 – Actions to execute when an outgoing Edge is Removed

D.5.4 Adding a Subnode

```
public void observe_tnode_subnodes(AddEvent ev) {
  ITNode ts = (ITNode)ev.new_value();
  //v.out_edges «mapped_to» tn.subnodes->iterate(ts,acc:Set|acc->includes(Pair{tn,ts}))
  if (manager().translate2(ts) == null) {
    IDirectedGraph graph = vertex().graph();
    IVertex v = new Vertex();
    IEdge edge = new Edge(vertex(),v);
    ((Translator)manager()).createMapping(v,ts);
    ((Translator)manager()).createMapping(edge,new Pair(tnode(),ts));
    graph.vertices().add(v);
    graph.edges().add(edge);
  }
}
```

Table 49 – Actions to execute when a Subnode is Added

D.5.5 Removing a Subnode

```
public void observe_tnode_subnodes(RemoveEvent ev) {
  ITNode ts = (ITNode)ev.old_value();
  //v.out_edges «mapped_to» tn.subnodes->iterate(ts,acc:Set|acc->includes(Pair{tn,ts}))
  IVertex v = (IVertex)manager().translate2(ts);
  if (v != null) {
    IDirectedGraph graph = v.graph();
    manager().removeMapping(v,ts);
    graph.vertices().remove(v);
  }
}
```

Table 50 – Actions to execute when a Subnode is Removed

D.6 Edge<->(TNode,TNode)

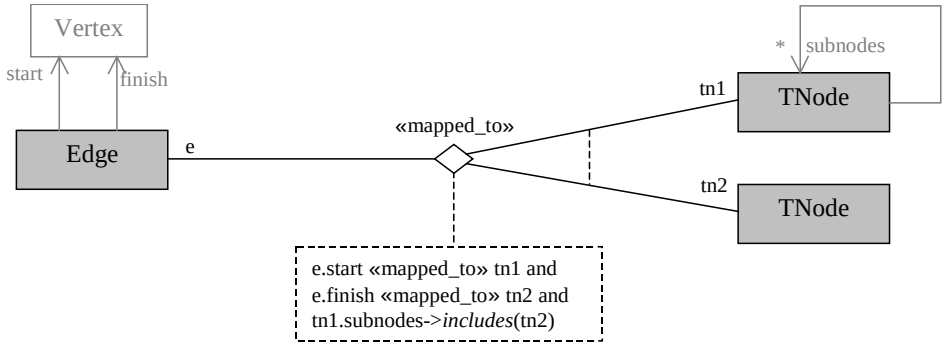

Figure 99 – Edge<->(TNode,TNode) mapping specification

D.6.1 Implementation Framework

```
public class Edge_PairTNode
  extends AbstractMapping
{
  private IEdge _edge;
  public IEdge edge() {return _edge;}

  private ITNode _tn1;
  public ITNode tn1() {return _tn1;}

  private ITNode _tn2;
  public ITNode tn2() {return _tn2;}

  public Edge_PairTNode(Iedge edge, IPair p) {
    _edge = edge;
    _tn1 = (ITNode)p.fst();
    _tn2 = (ITNode)p.snd();
    startObserving();
```

```
  }
  public void observe_edge_start(ChangeEvent ev) { ... }
  public void observe_edge_finish(ChangeEvent ev) { ... }
  public void observe_tn1(ChangeEvent ev) { ... }
  public void observe_tn2_parent(ChangeEvent ev) { ... }
  public void observe_edge(IObservableEvent e) {
    if (e.name().equals("start")) observe_edge_start((ChangeEvent)e);
    if (e.name().equals("finish")) observe_edge_finish((ChangeEvent)e);
  }
  public void observe_tn1_subnodes(IObservableEvent e) {
    if (e instanceof AddEvent) observe_tn1_subnodes((AddEvent)e);
    if (e instanceof RemoveEvent) observe_tn1_subnodes((RemoveEvent)e);
  }
  public void observe_tn2(IObservableEvent e) {
    if (e.name().equals("parent")) observe_tn2_parent((ChangeEvent)e);
  }
  //--- AbstractMapping ---
  public Object object1() {return _edge;}
  public Object object2() {return new Pair(_tn1,_tn2); }
  //--- Observer ---
  public void startObserving() {
    ((IObservable)_edge).addObserver(this);
    ((IObservable)_tn1.subnodes()).addObserver(this);
    ((IObservable)_tn2).addObserver(this);
  }
  public void stopObserving() {
    ((IObservable)_edge).removeObserver(this);
    ((IObservable)_tn1.subnodes()).removeObserver(this);
    ((IObservable)_tn2).removeObserver(this);
  }
  public void observe(IobservableEvent e) {
    if (e.source()==edge()) observe_edge(e);
    if (e.source()==tn1().subnodes()) observe_tn1_subnodes(e);
    if (e.source()==tn2()) observe_tn2(e);
  }
}
```

Table 51 – Implementation Framework for Edge↔(TNode,TNode) Mapping Class

D.6.2 Changing the start Attribute

```
  public void observe_edge_start(ChangeEvent ev) {
    IVertex old_v = (IVertex)ev.old_value();
    IVertex new_v = (IVertex)ev.new_value();
    ITNode new_tn1 = (ITNode)manager().translate1(new_v);
    if (new_tn1 != tn1()) {
      //e.start <-> tn1 and
      manager().removeMapping(edge(),new Pair(tn1(),tn2()));
      ((Translator)manager()).createMapping(edge(),new Pair(new_tn1,tn2()));
      //e.finish <-> tn2 and
      // nothing
      // tn1.subnodes->includes(tn2);
      tn1().subnodes().remove(tn2());
      new_tn1.subnodes().add(tn2());
      _tn1 = new_tn1;
    }
  }
```

Table 52 – Actions to execute when the start Attribute is Changed

D.6.3 Changing the finish Attribute

```
  public void observe_edge_finish(ChangeEvent ev) {
    IVertex old_v = (IVertex)ev.old_value();
```

```
    IVertex new_v = (IVertex)ev.new_value();
    ITNode new_tn2 = (ITNode)manager().translate1(new_v);
    if (new_tn2 != tn2()) {
      //e.start <-> tn1 and
      // nothing
      //e.finish <-> tn2 and
      manager().removeMapping(edge(),new Pair(tn1(),tn2()));
      ((Translator)manager()).createMapping(edge(),new Pair(tn1(),new_tn2));
      // tn1.subnodes->includes(tn2);
      tn1().subnodes().remove(tn2());
      tn1().subnodes().add(new_tn2);
      _tn2 = new_tn2;
    }
  }
```

Table 53 – Actions to execute when the finish Attribute is Changed

D.6.4 Adding a Subnode to tn1

Actions for adding subnodes are handled by the Vertex↔TNode mapping.

```
  public void observe_tn1(AddEvent ev) {
    //e.start <-> tn1 and
    // nothing
    //e.finish <-> tn2 and
    // nothing
    // tn1.subnodes->includes(tn2);
    // nothing
  }
```

Table 54 – Actions to execute when a Subnode is Added

D.6.5 Removing a Subnode from tn1

Actions for removing subnodes are handled by the Vertex↔TNode mapping.

```
  public void observe_tn1(RemoveEvent ev) {
    //e.start <-> tn1 and
    // nothing
    //e.finish <-> tn2 and
    // nothing
    // tn1.subnodes->includes(tn2);
    // nothing
  }
```

Table 55 – Actions to execute when a Subnode is Removed

D.6.6 Changing the parent of Subnode tn2

```
  public void observe_tn2_parent(ChangeEvent ce) {
    ITNode new_parent = (ITNode)ce.new_value();
    //e.start <-> tn1 and
    if (new_parent != tn1()) {
      IVertex v = (IVertex)manager().translate2(new_parent);
      edge().setStart(v);
      _tn1 = new_parent;
    }
    //e.finish <-> tn2 and
    // nothing
    // tn1.subnodes->includes(tn2);
    // nothing
  }
```

Table 56 – Actions to execute when a Subnode is Removed

Appendix E

Java-to-Tree Translator

This appendix illustrates the Java↔Tree translator, including all of the automatically generated code (with manual additions) for an active translator implementation.

E.1 Specification

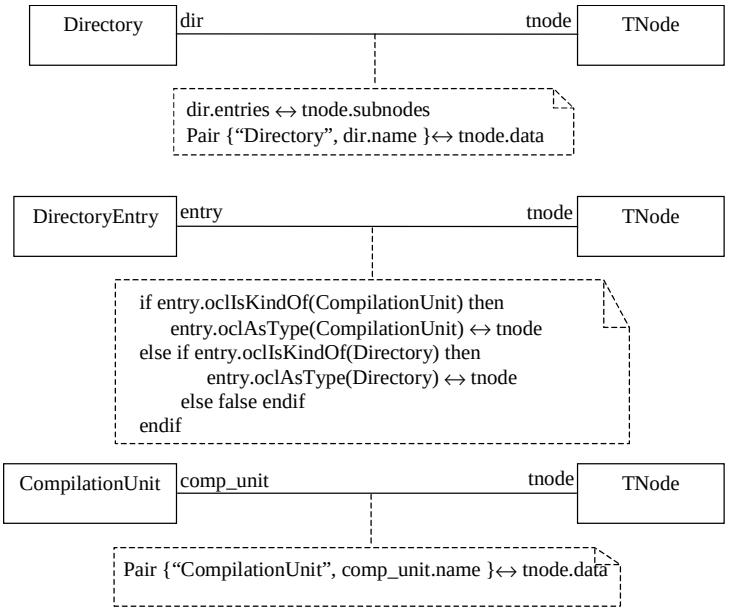

Figure 100 – Java ↔ Tree Mapping Specifications

E.2 ‘mappings’ Package

```
public class IDirectory_ITNode
  extends OclAny
{
  private IDirectory _dir;
  public IDirectory dir() {return _dir;}

  private ITNode _tnode;
  public ITNode tnode() {return _tnode;}

  public IOclConstraint constraint =
   OCL.Invarient(this,
     "self.dir.entries->size = self.tnode.subnodes->size "+
     "and self.dir.name = self.tnode.data.oclAsType('ukc.dha.utils.Pair').snd");

  public IDirectory_ITNode(IDirectory dir, ITNode tnode) {
    _dir=dir;
    _tnode=tnode;
  }
}
```

Table 57 – Directory↔TNode Mapping Class

```
public class IDirectoryEntry_ITNode
  extends OclAny
{
  private IDirectoryEntry _entry;
  public IDirectoryEntry entry() {return _entry;}

  private ITNode _tnode;
  public ITNode tnode() {return _tnode;}

  public IOclConstraint constraint = OCL.Invarient(this,"true");

  public IDirectoryEntry_ITNode(IDirectoryEntry entry, ITNode tnode) {
    _entry=entry;
    _tnode=tnode;
  }
}
```

Table 58 – DirectoryEntry↔TNode Mapping Class (Unused)

```
public class ICompilationUnit_ITNode
  extends OclAny
{
  private ICompilationUnit _comp_unit;
  public ICompilationUnit comp_unit() {return _comp_unit;}

  private ITNode _tnode;
  public ITNode tnode() {return _tnode;}

  public IOclConstraint constraint =
   OCL.Invarient(this,
   "self.comp_unit.declarations->size = self.tnode.subnodes->size and"+
   "self.comp_unit.name = self.tnode.data.oclAsType('ukc.dha.utils.Pair').snd");

  public ICompilationUnit_ITNode(ICompilationUnit comp_unit, ITNode tnode) {
    _comp_unit=comp_unit;
    _tnode=tnode;
  }
}
```

Table 59 – CompilationUnit↔TNode Mapping Class

E.3 'translator' Package

```
public class ConsistencyManager
  extends MappingManager
  implements IConsistencyManager
{
  ITranslator _trans;
  public void setTranslator(ITranslator t) {_trans=t;}


  public void createMapping(IDirectoryEntry entry, ITNode tnode) {
    new IDirectoryEntry_ITNode(entry, tnode,_trans);
    mappings().put(entry, tnode);
  }

  public void createMapping(IDirectory dir, ITNode tnode) {
    new IDirectory_ITNode(dir, tnode,_trans);
    mappings().put(dir, tnode);
  }

  public void createMapping(ICompilationUnit comp_unit, ITNode tnode) {
    new ICompilationUnit_ITNode(comp_unit, tnode,_trans);
    mappings().put(comp_unit, tnode);
  }

  //--- IConsistencyManager ---
  boolean _observing = false;
  public boolean is_observing() { return _observing; }
  public void is_observing(boolean b) { _observing=b; }
}
```

Table 60 – ConsistencyManager Class

```
public class Translator
  extends AbstractTranslator
```

```
{
  public Translator(IConsistencyManager cm,
                    IGenerator gen1To2,
                    IGenerator gen2To1) {
    super(cm, gen1To2, gen2To1);
  }

  public Object translate1To2(Object obj) {
    if (obj instanceof IDirectoryEntry)
      return translateIDirectoryEntryToITNode((IDirectoryEntry)obj);
    if (obj instanceof IDirectory)
      return translateIDirectoryToITNode((IDirectory)obj);
    if (obj instanceof ICompilationUnit)
      return translateICompilationUnitToITNode((ICompilationUnit)obj);
    if (obj instanceof IClassDeclaration)
      return translateIClassDeclarationToITNode((IClassDeclaration)obj);
    throw new RuntimeException(
      "Error:: Type "+obj.getClass()+" not handled in "
      +this.getClass().getName()+".translate1To2");
  }

  public Object translate2To1(Object obj) {
    if (obj instanceof ITNode)
      return translateITNodeToIDirectoryEntry((ITNode)obj);
    if (obj instanceof ITNode)
      return translateITNodeToIDirectory((ITNode)obj);
    if (obj instanceof ITNode)
      return translateITNodeToICompilationUnit((ITNode)obj);
    if (obj instanceof ITNode)
      return translateITNodeToIClassDeclaration((ITNode)obj);
    throw new RuntimeException(
      "Error:: Type "+obj.getClass()+" not handled in "
      +this.getClass().getName()+".translate2To1");
  }

  public ITNode translateIDirectoryEntryToITNode(IDirectoryEntry component1) {
    ITNode component2 =
            (ITNode)mapping_manager().mappings().get1To2(component1);
    if (component2 == null) {
      return (ITNode)create1To2(ITNode.class,IDirectoryEntry.class,component1);
    }
    return component2;
  }

  public ITNode translateIDirectoryToITNode(IDirectory component1) {
    ITNode component2 =
            (ITNode)mapping_manager().mappings().get1To2(component1);
    if (component2 == null) {
      return (ITNode)create1To2(ITNode.class,IDirectory.class,component1);
    }
    return component2;
  }

  public ITNode translateICompilationUnitToITNode(ICompilationUnit component1) {
    ITNode component2 =
            (ITNode)mapping_manager().mappings().get1To2(component1);
    if (component2 == null) {
      return (ITNode)create1To2(ITNode.class,ICompilationUnit.class,component1);
    }
    return component2;
  }

  public IDirectoryEntry translateITNodeToIDirectoryEntry(ITNode component2) {
    IDirectoryEntry component1 =
            (IDirectoryEntry)mapping_manager().mappings().get2To1(component2);
    if (component1 == null) {
      return (IDirectoryEntry)
             create2To1(IDirectoryEntry.class,ITNode.class,component2);
    }
    return component1;
  }

  public IDirectory translateITNodeToIDirectory(ITNode component2) {
    IDirectory component1 =
            (IDirectory)mapping_manager().mappings().get2To1(component2);
    if (component1 == null) {
      return (IDirectory)create2To1(IDirectory.class,ITNode.class,component2);
```

```
    }
    return component1;
  }

  public ICompilationUnit translateITNodeToICompilationUnit(ITNode component2) {
    ICompilationUnit component1 =
            (ICompilationUnit)mapping_manager().mappings().get2To1(component2);
    if (component1 == null) {
      return (ICompilationUnit)
             create2To1(ICompilationUnit.class,ITNode.class,component2);
    }
    return component1;
  }
}
```

Table 61 – Translator Class

```
public class JavaGenerator
  extends AbstractGenerator
{
  IJavaBuilder _builder;

  private ConsistencyManager _mappings;
  public ConsistencyManager consistency_manager() { return _mappings; }

  public JavaGenerator(IJavaBuilder builder, IMappingManager mm) {
    _builder = builder;
    _mappings = (ConsistencyManager)mm;
  }

  public IDirectoryEntry createIDirectoryEntry( ITNode tn ) {
    Pair p = (Pair)tn.data();
    String s = ((IString)p.fst()).toString();
    if (s.equals("Directory")) return createIDirectory(tn);
    if (s.equals("CompilationUnit")) return createICompilationUnit(tn);
    throw new RuntimeException("Error:: Unknown TNode type - "+s);
  }

  public IDirectory createIDirectory( ITNode tn ) {
    IDirectory d = (IDirectory)consistency_manager().get2To1(tn.parent());
    if (d != null) {
      IDirectory subd = _builder.buildDirectory(d);
      subd.name().setTo((IString)((Pair)tn.data()).fst();
      consistency_manager().createMapping(subd,tn);
      return subd;
    }
    throw new RuntimeException("Error:: No mapping for parent of "+tn);
  }

  public ICompilationUnit createICompilationUnit( ITNode tn ) {
    IDirectory d = (IDirectory)consistency_manager().get2To1(tn.parent());
    if (d != null) {
      ICompilationUnit cu = _builder.buildCompilationUnit(d);
      cu.name().setTo((IString)((Pair)tn.data()).fst();
      consistency_manager().createMapping(cu,tn);
      return cu;
    }
    throw new RuntimeException("Error:: No mapping for parent of "+tn);
  }

}
```

Table 62 – Java Generator Class

```
public class TreeGenerator
  extends AbstractGenerator
{
    ITreeBuilder _builder;

    private ConsistencyManager _mappings;
    public ConsistencyManager consistency_manager() { return _mappings; }

    public TreeGenerator(ITreeBuilder builder, IConsistencyManager mm) {
      _builder = builder;
      _mappings = (ConsistencyManager)mm;
    }

    public ITNode createITNode( IDirectoryEntry de  ) {
```

```
       if (de instanceof IDirectory) return createITNode((IDirectory)de);
       if (de instanceof ICompilationUnit)
            return createITNode((ICompilationUnit)de);
       throw new RuntimeException("Error:: Unknown subtype of IDirectoryEntry");
    }

    public ITNode createITNode( IDirectory d ) {
       ITNode pn = (ITNode)consistency_manager().get1To2(d.parent());
       if (pn != null) {
           ITNode tn =  _builder.buildTNode(pn);
           tn.setData( new Pair(OCL.String("Directory"),d.name()) );
           consistency_manager().createMapping(d,tn);
           return tn;
       }
       throw new RuntimeException("Error:: Can't create TNode
                                           for a root directory.");
    }

    public ITNode createITNode( ICompilationUnit cu ) {
       ITNode pn = (ITNode)consistency_manager().get1To2(cu.parent());
       if (pn != null) {
           ITNode tn = _builder.buildTNode(pn);
           tn.setData( new Pair(OCL.String("CompilationUnit"),cu.name()) );
           consistency_manager().createMapping(cu,tn);
           return tn;
       }
       throw new RuntimeException("Error:: No mapping for parent of "+cu);
    }
}
```

Table 63 – Tree Generator Class

Appendix F

SVG-to-Graph-to-Automaton Translator

This Appendix contains the specification of the SVG↔Graph↔Automaton Translator for the Stochastic Automaton Editor. It also includes the XMI encoding of the translator specifications, used as input to the automatic implementation generator.

F.1 Specifications

F.1.1 SVG↔DirectedGraph

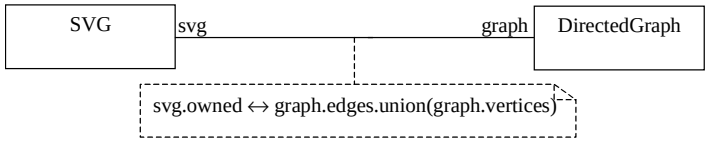

Figure 101 – SVG↔DirectedGraph Mapping Specification

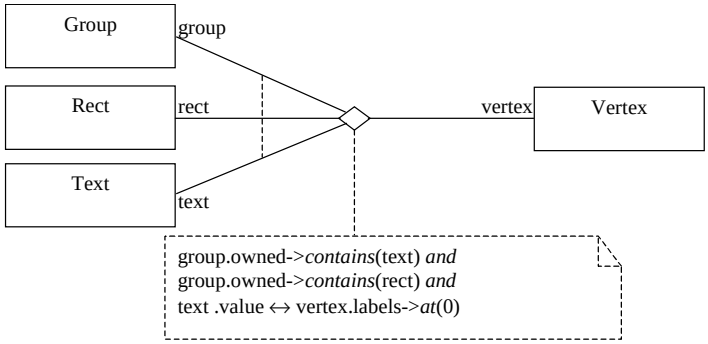

Figure 102 – (Group,Rect,Text) ↔ Vertex Mapping Specification

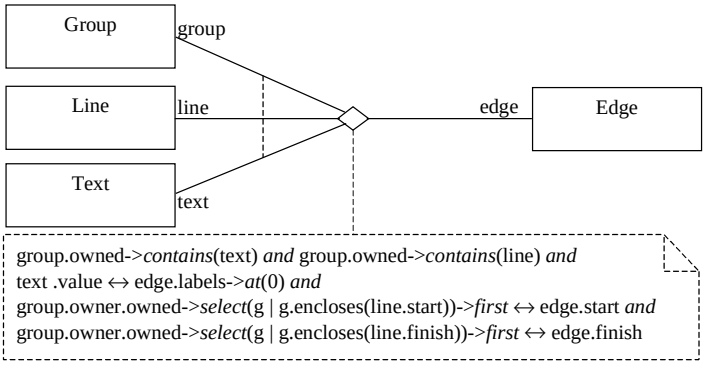

Figure 103 – (Group,Line,Text) ↔ Vertex Mapping Specification

F.1.2 DirectedGraph↔Automaton

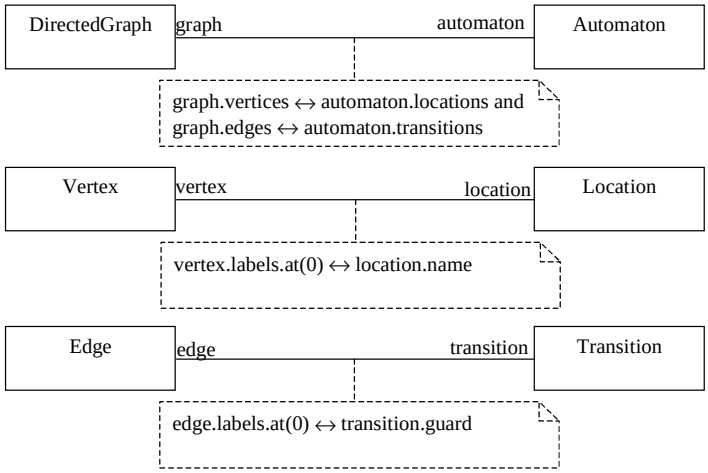

Figure 104 – DirectedGraph↔Automata Translator Specification

F.2 XMI

```
<XMI version="1.1" xmlns:UML="org.omg/UML1.3">
  <XMI.header>
    <XMI.model xmi.name="SVG_Graph" href="SVG_Graph.xmi"/>
    <XMI.import xmi.name="svg" href="SVG.xmi"/>
    <XMI.import xmi.name="directedGraph" href="DirectedGraph.xmi"/>
  </XMI.header>
  <XMI.content>
    <UML:Association>
      <UML:Association.connection>
        <UML:AssociationEnd name="svg" xmi.id="svg" >
          <UML:AssociationEnd.type><UML:Classifier name="SVG" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="graph" xmi.id="graph" >
          <UML:AssociationEnd.type><UML:Classifier name="DirectedGraph" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
          <UML:Constraint.body xmi.value=
             "self.graph.vertices->union(self.graph.edges)->size
                =
              self.svg.owned->size" />
          </UML:Constraint>
      </UML:ModelElement.constraint>
    </UML:Association>

    <UML:Association>
      <UML:Association.connection>
        <UML:AssociationEnd name="group" xmi.id="group_gv" >
          <UML:AssociationEnd.type><UML:Classifier name="Group" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="vertex" xmi.id="vertex_gv" >
          <UML:AssociationEnd.type><UML:Classifier name="Vertex" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
          <UML:Constraint.body xmi.value="true" />
        </UML:Constraint>
      </UML:ModelElement.constraint>
    </UML:Association>

    <UML:Association>
      <UML:Association.connection>
```

```
        <UML:AssociationEnd name="group" xmi.id="group_grtv" >
          <UML:AssociationEnd.type><UML:Classifier name="Group" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="rect" xmi.id="rect_grtv" >
          <UML:AssociationEnd.type><UML:Classifier name="Rect" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="text" xmi.id="text_grtv" >
          <UML:AssociationEnd.type><UML:Classifier name="Text" />
                                       </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="vertex" xmi.id="vertex_grtv" >
          <UML:AssociationEnd.type><UML:Classifier name="Vertex" />
                                         </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
          <UML:Constraint.body xmi.value=
                   "self.group.owned->contains(text) and
                    self.group.owned->contains(rect) and
                    self.text.value=self.vertex.labels->at(0)" />
        </UML:Constraint>
      </UML:ModelElement.constraint>
      <UML:ModelElement.ties><UML:Tie xmi.idref="tie_a1" />
                                       </UML:ModelElement.ties>
      <UML:ModelElement.ties><UML:Tie xmi.idref="tie_a2" />
                                       </UML:ModelElement.ties>
    </UML:Association>

    <UML:Tie xmi.id="tie_a1" >
      <UML:Tie.tied_elements>
        <UML:AssociationEnd xmi.idref="group_grtv" />
        <UML:AssociationEnd xmi.idref="rect_grtv" />
        <UML:AssociationEnd xmi.idref="text_grtv" />
      </UML:Tie.tied_elements>
    </UML:Tie>

    <UML:Tie xmi.id="tie_a2" >
      <UML:Tie.tied_elements>
        <UML:AssociationEnd xmi.idref="vertex_grtv" />
      </UML:Tie.tied_elements>
    </UML:Tie>

    <UML:Association>
      <UML:Association.connection>
        <UML:AssociationEnd name="group" xmi.id="group_glte" >
          <UML:AssociationEnd.type><UML:Classifier name="Group" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="line" xmi.id="line_glte" >
          <UML:AssociationEnd.type><UML:Classifier name="Line" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="text" xmi.id="text_glte" >
          <UML:AssociationEnd.type><UML:Classifier name="Text" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="edge" xmi.id="edge_glte" >
          <UML:AssociationEnd.type><UML:Classifier name="Edge" />
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
          <UML:Constraint.body xmi.value=
                   "self.group.owned->contains(text) and
                    self.group.owned->contains(line) and
                    self.text.value=self.edge.labels->at(0) " />
        </UML:Constraint>
      </UML:ModelElement.constraint>

      <UML:ModelElement.ties><UML:Tie xmi.idref="tie_b1" />
                                       </UML:ModelElement.ties>
      <UML:ModelElement.ties><UML:Tie xmi.idref="tie_b2" />
                                       </UML:ModelElement.ties>
```

```
    </UML:Association>

    <UML:Tie xmi.id="tie_b1" >
      <UML:Tie.tied_elements>
        <UML:AssociationEnd xmi.idref="group_glte" />
        <UML:AssociationEnd xmi.idref="line_glte" />
        <UML:AssociationEnd xmi.idref="text_glte" />
      </UML:Tie.tied_elements>
    </UML:Tie>

    <UML:Tie xmi.id="tie_b2" >
      <UML:Tie.tied_elements>
        <UML:AssociationEnd xmi.idref="edge_glte" />
      </UML:Tie.tied_elements>
    </UML:Tie>

  </XMI.content>
</XMI>
```

Table 64 – XMI for SVG↔DirecteGraph

```
<XMI version="1.1" xmlns:UML="org.omg/UML1.3">
  <XMI.header>
    <XMI.model xmi.name="Graph_Automata" href="Graph_Automata.xml"/>
    <XMI.metamodel xmi.name="UML" href="UML.xml"/>
    <XMI.metamodel xmi.name="OCL" href="OCL.xml"/>
  </XMI.header>
  <XMI.content>

    <UML:Association>
      <UML:Association.connection>
        <UML:AssociationEnd name="graph" xmi.id="graph" >
          <UML:AssociationEnd.type><UML:Classifier name="DirectedGraph"/>
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="automaton" xmi.id="automaton" >
          <UML:AssociationEnd.type><UML:Classifier name="Automaton"/>
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
          <UML:Constraint.body xmi.value=
                 "self.graph.vertices->size = self.automaton.locations->size and
                  self.graph.edges->size = self.automaton.transitions->size"/>
        </UML:Constraint>
      </UML:ModelElement.constraint>
   </UML:Association>

    <UML:Association>
      <UML:Association.connection>
        <UML:AssociationEnd name="vertex" xmi.id="vertex" >
          <UML:AssociationEnd.type><UML:Classifier name="Vertex"/>
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="location" xmi.id="location" >
          <UML:AssociationEnd.type><UML:Classifier name="Location"/>
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
          <UML:Constraint.body xmi.value=
                   "self.vertex.labels.at(0) = self.location.name"/>
        </UML:Constraint>
      </UML:ModelElement.constraint>
   </UML:Association>

    <UML:Association>
      <UML:Association.connection>
        <UML:AssociationEnd name="edge" xmi.id="edge" >
          <UML:AssociationEnd.type><UML:Classifier name="Edge"/>
                                          </UML:AssociationEnd.type>
        </UML:AssociationEnd>
        <UML:AssociationEnd name="transition" xmi.id="transition" >
          <UML:AssociationEnd.type><UML:Classifier name="Transition"/>
```

```
                                              </UML:AssociationEnd.type>
        </UML:AssociationEnd>
      </UML:Association.connection>
      <UML:ModelElement.constraint>
        <UML:Constraint>
          <UML:Constraint.body xmi.value=
                    "self.edge.labels.at(0) = self.transition.guard"/>
        </UML:Constraint>
      </UML:ModelElement.constraint>
   </UML:Association>

  </XMI.content>
</XMI>
```

Table 65 – XMI for DirectedGraph↔Automaton

Appendix G

UML Actions-to-RiscSim

The example translator contained within this appendix illustrates a portion of a translator that could be used to map a UML model onto a RiscSim Petri-Net. Such a translator could be used as part of the Permabase toolkit for performance modelling, as described in Chapter 3.

G.1 UML Actions

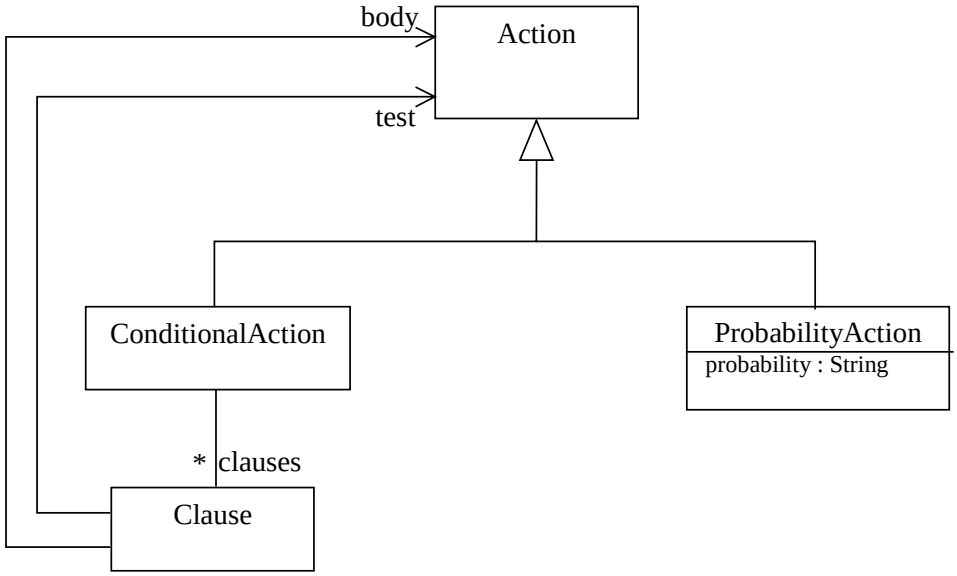

Figure 105 – A Partial UML Actions model

This segment of a UML Actions model is taken from [OMG_00aug]. The additional ProbabilityAction class is added to enable non-deterministic tests to be defined for condition clauses.

G.2 RiscSim

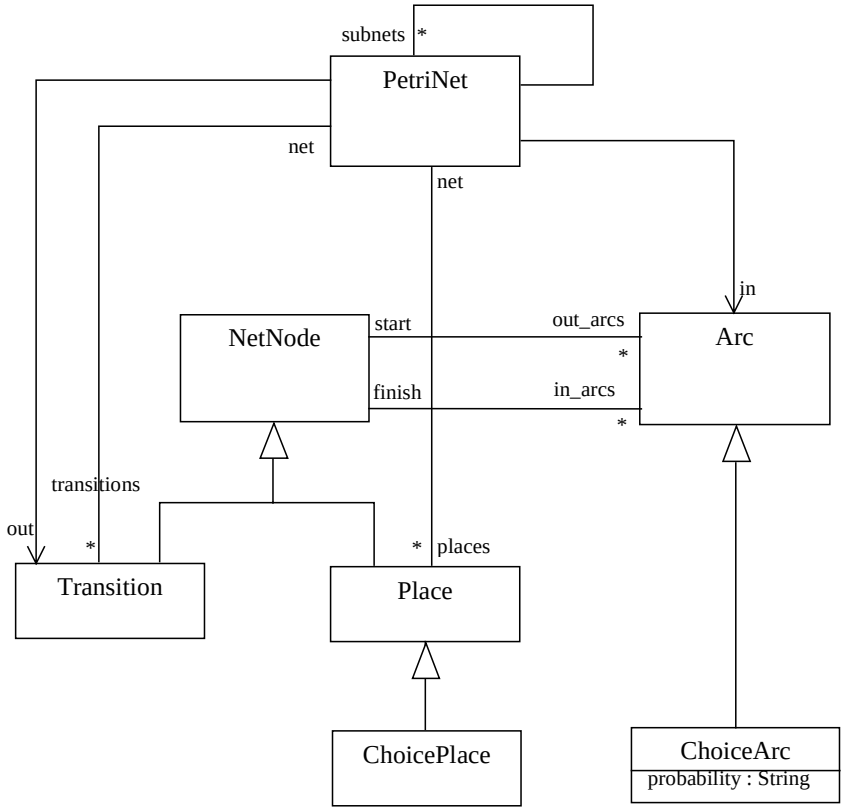

Figure 106 – Partial RiscSim Model

This model illustrates the components of a RiscSim model that are required for the example translation. The RiscSim tool is described in [Linington_99apr].

G.3 Translator Mappings

To form the mapping from Action to PetriNet, we map each Action to a sub-PetriNet that has a starting (in) Arc and a finishing (out) transition. The behaviour of a particular action is fully contained within the specified sub-PetriNet.

To generate a UMLAction model from a PetriNet model, a function would be required that searches for sub-PetriNets within a parent PetriNet detecting fully contained sub-nets. However, as the translator (for the purpose of Permabase) is to create a PetriNet model from a UMLAction model, the OCL *select* operation is sufficient; the set of ‘subnets’ will be created during the creation of the PetriNet model.

When implementing the generation code that creates PetriNets from Action objects, the correct pattern of net must be created. The constraints in the following specification do explicitly define the required pattern, but to aid the readers understanding of the patterns of net required, Figure 107 indicates the mapping between segments of PetriNet and pseudo code representation of the Actions.

In Figure 107, P marks ordinary Places, CP marks a ChoicePlace and CA marks a ChoiceArc.

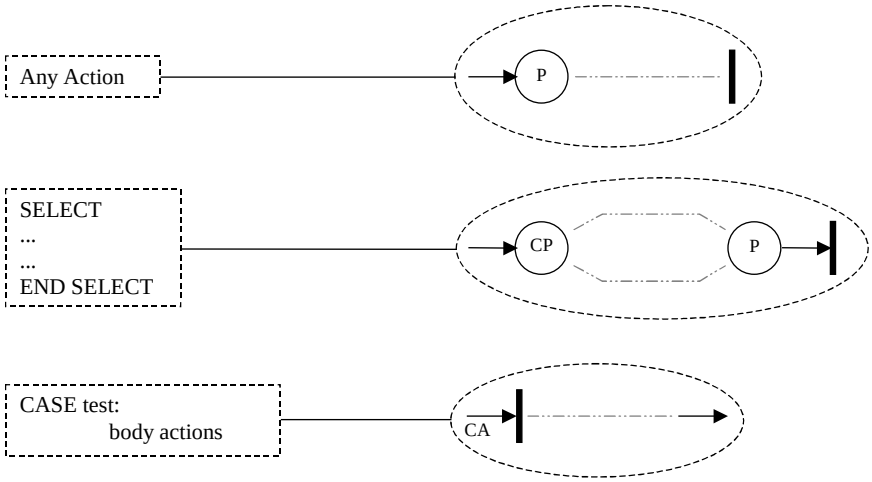

Figure 107 – Mapping Between Pseudo Code Actions and PetriNet Segments

All Actions are mapped to a pattern of PetriNet with a starting (in) Arc, and an ending (out) Transition. A Conditional Action (represented by the SELECT statement) maps to a sub-PetriNet with an initial ChoicePlace that branches out to the various sub-clauses in the condition. A ChoicePlace is a RiscSim specific style of PetriNet place that non-deterministically chooses one of the outgoing ChoiceArcs, based on the probabilities associated with them. The Clause of a conditional action (CASE statement) maps to a subnet starting with a ChoiceArc, which represents the 'test' and containing the body of the clause as a further subnet.

The functionality provided by the RiscSim engine limits the translation such that any test on a clause must be a ProbabilityAction. Other (deterministic) tests cannot be mapped to a RicSim model.

The mappings for the translator specification, shown in Figure 108, contain enough information to implement a one way translation from UMLActions to RiscSim model.

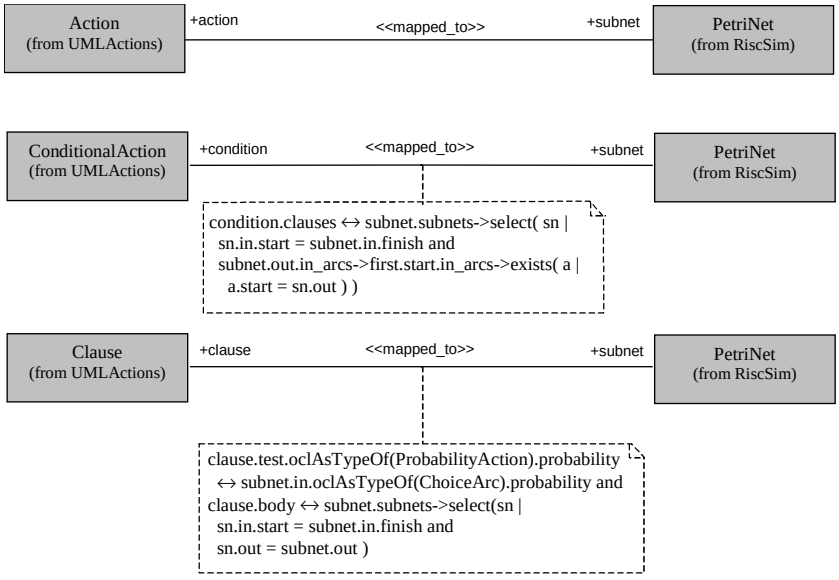

Figure 108 – Partial UMLActions↔RiscSim Translator Specification

Bibliography

[Abiteboul_etal_94] S. Abiteboul, M. Adiba, J. Arlow, P. Armenise, S. Bandinelli, L. Baresi, P. Breche, F. Buddrus, C. Collet, P. Corte, Th. Coupaye, C. Delobel, W. Emmerich, G. Ferran, F. Ferrandina, A. Fuggetta, C. Ghezzi, S.E. Lautemann, L. Lavazza, J. Madec, M. Phoenix, S. Sachweh, W. Schäfer, C. Souza dos Santos, G. Tigg and R. Zicari; *The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes;* Proc. of the 1st Asian Pacific Software Engineering Conf. Tokyo, Japan, IEEE Computer Society Press; 1994; 10-19.

[Adobe_98apr] Nabeel Al-Shamma, Robert Ayers, Richard Cohn, Jon Ferraiolo, Martin Newell, Roger K. de Bry, Kevin McCluskey, Jerry Evans; *Precision Graphics Markup Language (PGML);* Adobe Systems Incorporated; April 1998.

[Aho_etal_86] A. V. Aho, R. Sethi, J. D. Ullman; *Compilers: Principles, Techniques and Tools;* Addison-Wesley, ISBN 0201101947; 1986.

[Akehurst_00] David Akehurst; *An OO visual language definition approach supporting multiple views;* Proc. VL2000, IEEE Symposium on Visual Languages; September 2000.

[Akehurst_Bordbar_01] D.H.Akehurst, B.Bordbar; On Querying UML data models with OCL; Proceedings of <<UML>> 2001 "Modeling Languages, Concepts and Tools", Toronto, Ontario, Canada; to appear October 2001.

[Akehurst_etal_00] David Akehurst, Howard Bowman, Jeremy Bryans, John Derrick; *A Manual for a ModelChecker for Stochastic Automata;* University of Kent, Computing Laboratory, Technical Report 9-00; December 2000.

[Akehurst_Waters] D.H.Akehurst, A.G.Waters; *UML Deficiencies from the perspective of automatic Performance Model Generation;* OOPSLA '99 Workshop on Rigorous Modelling and Analysis with the UML: Challenges and Limitations; November 1999.

[Alexander_etal_77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel; *A Pattern Language;* Oxford University Press, New York; 1977.

[Bahlke_Snelting_86] Rolf Bahlke, Gregor Snelting; *The PSG system: from formal language definitions to interactive programming environments;* ACM Transactions on Programming Languages and Systems, Volume 8, Issue 4; 1986; 547-576.

[Bancilhon_etal_92] Francois Bancilhon, Claude Delobel, Paris Kanellakis; *Building an Object-Oriented Database System: The Story of O2;* Morgan Kaufmann Publishers; ISBN: 1558601694; May 1992.

[Bardohl_etal_99] R. Bardohl, G. Taentzer, M. Minas, A. Schürr; *Application of Graph Transformation to Visual Languages;* Handbook on Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools, World Scientific; 1999.

[Bézivin_etal_95] J. Bézivin, J. Lanneluc, R. Lemesle; *sNets: The Core Formalism for an Object-Oriented CASE Tool;* Object-Oriented Technology for Database and Software Systems, V.S. Algar & R. Missaoui ed., World Scientific Publishers; 1995; 224-239.

[Blaha_Premerlani_96] Michael Blaha and William Premerlani; *A Catalog of Object Model Transformations;* Presented at 3rd Working Conference on Reverse Engineering, Monterey, California; November 1996.

[Boiten_etal_95] Eerke Boiten, Howard Bowman, John Derrick, and Maarten Steen; *Cross viewpoint consistency in open distributed processing (intra language consistency);* Technical Report 8-95, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK; June 1995.

[Bollobás_79] Béla Bollobás; *Graph Theory An Introductory Course;* Springer-Verlag; 1979.

[Booch_etal] Grady Booch, James Rumbaugh, Ivar Jacobson; *The Unified Modeling Language User Guide;* Addison Wesley Longman Inc.; 1999.

[Bordbar_etal_01] B. Bordbar, J. Derrick, G. Waters; *Using UML to specify QoS constraints in ODP;* IEEE Computer Networks; under review.

[Borras_etal_88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang and V. Pascual; *Centaur: the system;* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical software development environments, Boston, MA USA; November 1988; 14-24.

[Bowman_etal_96jan] Howard Bowman, John Derrick, Peter Linington, Maarten Steen; *Cross-viewpoint consistency in Open Distributed Processing;* Software Engineering Journal, Vol. 11, No. 1; January 1996; 44-57.

[Breu_etal97] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, Veronika Thurner; *Towards a Formalization of the Unified Modeling Language;* Proceedings of ECOOP'97; 1997.

[Britton_Jones_99] Carol Britton and Sara Jones; *The Untrained Eye: How Languages for Software Specification Support Understanding in Untrained Users;* HCI Volume 14 (1); 1999; 191-244.

[Bryans_etal_00jan] J.W. Bryans, H. Bowman and J. Derrick; *A model checking algorithm for stochastic systems;* Technical Report 4-00, University of Kent at Canterbury, Canterbury, Kent; January 2000.

[Bryans_etal_00nov] Jeremy Bryans, Lynne Blair, Howard Bowman; *Specification and analysis of automata-based designs;* Integrated Formal Methods (IFM 2000), volume 1945 of Lecture Notes in Computer Science; November 2000; 176-193.

[Bryans_etal_99nov] J.W. Bryans, H. Bowman and J. Derrick; *Stochastic Model-Checking for Multimedia;* PONMS'99 (Modal and Temporal Logic Based Planning for Open Networked Multimedia Systems), Cape Cod, Massachusetts; November 1999.

[Burnett_etal_00] M. Burnett, N. Cao, J. Atwood; *Time in grid-oriented VPLs: just another dimension?;* IEEE International Symposium on Visual Languages (VL2000), Los Alamitos, CA: IEEE Computer Society; 2000; 137-144.

[Celentano_78] A. Celentano; *Incremental Parsers;* Acta Informatica, Volume 10; 1978; 307.

[Civello_93] Franco Civello; *Roles for Composite Objects in Object-Oriented Analysis and Design;* OOPSLA'93 Conference Proceedings, ACM SIGPLAN Notices, Vol. 28, N.10; October 1993.

[Claus_etal_78] edited by V. Claus, H. Ehrig, G. Rozenberg; *Proc. Int. Workshop on Graph Grammars and Their Applications to Computer Science And Biology, LNCS 73;* Berlin: Springer Verlag; 1978.

[Cook_etal_99] Steve Cook, Anneke Kleppe, Richard Mitchell, Jos Warmer, Alan Wills, Joaquin Miller, Rebecca Wirfs-Brock; *Defining the Context of OCL Expressions;* Proc. «UML» '99 The Unified Modelling Language: Beyond the Standard, Springer; October 1999; 372-383.

[Cook_Newson_96] Vivian Cook, Mark Newson; *Chompsky's Universal Grammar (2nd ed.);* Blackwell Publishers; ISBN: 0631195564; 1996.

[Dahm_99] Markus Dahm; *Byte Code Engineering;* Proceedings JIT'99; 1999; 267-277.

[DeRemer_74:1] F. L. DeRemer; *Review of Formalisms and Notation;* Compiler Construction: An Advanced Course, LNCS 21, ISBN 3540069585; 1974; 37-56.

[DeRemer_74:3] F. L. DeRemer; *Transformational Grammars;* Compiler Construction: An Advanced Course, LNCS 21, ISBN 3540069585; 1974; 121-145.

[Donnelly_Stallman_00] Charles Donnelly, Richard M. Stallman; *Bison: The YACC-Compatible Parser Generator;* iUniverse.Com, Inc.; ISBN: 0595100325; August 2000.

[Donzeau-Gouge_etal_84] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Mélèse; *Document structure and Modularity in Mentor;* Proceedings of the ACM SIGSOFT/SIGPLAN - Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Software Engineering Notes, Volume 9, No 3; 1984; 141-148.

[Dsouza_01mar] Desmond Dsouza; *Model-Driven Architecture and Integration: Opportunities and Challenges, Version 1.1;* Kinetium; March 2001.

[Dustzadeh_Najm_97] Joubine Dustzadeh, Elie Najm; *Consistent Semantics for ODP Information and Computational Models;* Proceedings of FORTE/PSTV'97; 1997.

[Emmerich_95] W. Emmerich; *Tool Construction for Process-Centred Software Development Environments based on Object Databases;* PhD Thesis, Dept. of Mathematics and Computer Science, University of Paderborn, Germany; 1995.

[Emmerich_96] Wolfgang Emmerich; *Tool Specification with GTSL;* Proc. of the 8th Int. Workshop on Software Specification and Design, Schloss Velen, Germany, IEEE Computer Society Press; 1996; 26-35.

[Emmerich_etal_93dexa] W. Emmerich, P. Kroha and W. Schäfer; *Object-oriented Database Management Systems for Construction of CASE Environments;* Database and Expert Systems Applications - Proc. of the 4th Int. Conf. DEXA '93, Prague, Czech Republic, LNCS 720; 1993; 631-642.

[Emmerich_etal_93esec] W. Emmerich, W. Schäfer and J. Welsh; *Databases for Software Engineering Environments - The Goal has not yet been attained;* Software Engineering ESEC '93 - Proc. of the 4th European Software Engineering Conference, Garmisch-Partenkirchen, Germany, LNCS 717; 1993; 145-162.

[Emmerich_etal_97] Wolfgang Emmerich, J. Arlow, J. Madec and M. Phoenix; *Tool Construction for the British Airways SEE with the O2 ODBMS;* Theory and Practice of Object Systems, Volume 3, Issue 3; 1997; 213-231.

[Engels_etal_86] G Engels, M Nagl and W Schafer; *On the structure of structure-oriented editors for different applications;* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on

Practical software development environments; December 1986; 190-198.

[Engels_etal_92] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer and A. Schürr; *Building integrated software development environments. Part I tool specification;* ACM Transactions on Software Engineering and Methodology, Volume 1, Issue 2; 1992; 135-167.

[Evans_etal98jun] A. Evans, R. France, K. Lano, B. Rumpe; *Developing the UML as a Formal Modelling Notation;* «UML» '98 Beyond the Notation; June 1998; 297-308.

[Feather_96] M. Feather; *Modularised Exception Handling;* Viewpoints 96: An International Workshop on Multiple Perspectives in Software Development, Joint Proc. of the SIGSOFT'96 Workshops, ACM Press, San Francisco; October 1996; 167-171.

[Finkelstein_etal_92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, M. Goedicke; *Viewpoints: A Framework for Integrating Multiple Perspectives in System Development;* International Journal of Software Engineering and Knowledge Engineering 2(1); March 1992; 31-58.

[Finkelstein_etal_94] Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B.; *Inconsistency Handling In Multi-Perspective Specifications;* IEEE Transactions on Software Engineering, Volume 20, Issue 8; August 1994; 569-578.

[Fischer_etal_99] T.Fischer, J.Niere, L.Torunski, A.Zündorf; *Story Diagrams: A new Graph Grammar Language based in the Unified Modeling Language;* Proc. of TAGT '98 - 6th International Workshop on Theory and Application of Graph Transformation. Technical Report tr-ri-98-201, University of Paderborn; 1999.

[Fowler_Scott] Martin Fowler with Kendall Scott; *UML Distilled: Applying the Standard Object Modeling Language;* Addison Wesley Longman, Inc.; 1997.

[Gamma_etal_94] Erich Gamma, Richard Helm, Ralph Johnson, John Vissides; *Design Patterns;* Addison Wesley; ISBN: 0201633612; December 1994.

[Garlan_Miller_84] D.B. Garlan and P.L. Miller; *GNOME: An Introductory Programming Environment Based on a Family of Structure Editors;* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments; April 1984.

[Ghezzi_Mandrioli_79] Carlo Ghezzi, Dino Mandrioli; *Incremental Parsing;* ACM Transactions on Programming Languages and Systems, Vol. 1; 1979; 58 - 70.

[Ghezzi_Mandrioli_80] Carlo Ghezzi, Dino Mandrioli; *Augmenting Parsers to Support Incrementality;* Journal of the ACM, Volume 27 Issue 3; July 1980; 564-579.

[Gil_etal_99] J Gil, J Howse, S Kent; *Constraint Diagrams: A Step Beyond UML;* Proceedings of TOOLS USA'99. IEEE Computer Society Press; December 1999.

[Gogolla_Presicce] Martin Gogolla, Francesco Parisi Presicce; *State Diagrams in UML: A Formal Semantics using Graph Transformations;* Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques; 1998.

[Gray_etal_92] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane and William M. Waite; *Eli: a complete, flexible compiler*

construction system; Communications of the ACM, Volume 35, Issue 2; 1992; 121-130.

[Green_89] T.R.G. Green; *Cognitive dimensions of notations;* People and Computers V, Cambridge University Press; 1989.

[Green_Petre_96] T. R. G. Green and M. Petre; *Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework;* Journal of Visual Languages and Computing, Vol. 7; 1996; 131-174.

[GSmith_99oct] Graeme Smith; *The Object-Z Specification Language;* Kluwer Academic Publishers; ISBN: 0792386841; October 1999.

[Habermann_Notkin_96] A. N. Habermann, David Notkin; *Gandalf Software Development Environments;* IEEE Transactions on Software Engineering Volume 12, Issue 12; December 1986; 1117-1127.

[Harel_87] David Harel; *Statecharts: A Visual Formalism for Complex Systems;* Science of Computer Programming, Vol. 8; 1987; 231-274.

[Harold_99jul] Elliotte Rusty Harold; *XML Bible;* IDG Books Worldwide, ISBN: 0764532367; July 1999.

[Harwood] Robin Harwood; *Object Oriented Development Method;* British Telecom; 1998.

[Hoffmann_99] Berthold Hoffmann; *From Graph Transformation Rules to Rule-based Visual Object-oriented Programs;* Proceedings of International Workshop and Symposium, AGTIVE - Applications of Graph Transformation with Industrial Relevance; September 1999.

[H-Sellers_Barbier_99] Brian Henderson-Sellers, Franck Barbier; *Black and White Diamonds;* Proc. «UML» '99 The Unified Modelling Language: Beyond the Standard, Springer; October 1999; 550-565.

[Hussmann_etal_00] Heinrich Hussmann; Birgit Demuth, Frank Finger; *Modular Architecture for a Toolset Supporting OCL;* <<UML>> 2000; October 2000.

[ISO/IEC_95:1] ISO/IEC; *Open Distributed Processing - Reference Model - Part 1: Overview;* International Standard 10746-1, ITU Recommendation X.901; July 1995.

[ISO/IEC_96] ISO/IEC 14977:1996; *Information technology -- Syntactic metalanguage -- Extended BNF;* ISO/IEC Copyright Office, Geneva.; 1996.

[Jahnke_etal_96] J.-H. Jahnke, W. Schäfer, and A. Zündorf; *A Design Environment for Migrating Relational to Object Oriented Database Systems;* Proc. of the International Conference on Software Maintenance (ICSM '96), IEEE Computer Society; 1996.

[Jahnke_Zündorf_98] J. H. Jahnke, A. Zündorf; *Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment;* Proc. of TAGT '98 - 6th International Workshop on Theory and Application of Graph Transformations. Technical Report, tr-ri-98-201, Paderborn; November 1998.

[Jahnke_Zündorf_99] J.-H. Jahnke and A. Zündorf; *Applying Graph Transformations To Database Re-Engineering;* Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2 - Applications, languages and tools; 1999.

[Jalili_Gallier_82] F. Jalili and J. H. Gallier; *Building friendly parsers;* Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages; January 1982; 196-206.

[Johnson_Fischer_82] G.F Johnson, C.N. Fischer; *Non-syntactic attribute flow in language based editors;* Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages; January 1982; 185-195.

[Kastens_80] U. Kastens; *Ordered Attributed Grammars;* Acta Informatica, Volume 13, Issue 3; 1980; 229-256.

[Kastens_Waite_94] U. Kastens, W. M. Waite; *Modularity and reusability in attribute grammars;* Acta Informatica, Volume 31; 1994; 601-627.

[Keller_etal_84] S. E. Keller, J. A. Perkins, T. F. Payton, S. P. Mardinly; *Tree Transformation Techniques and Experiences;* SIGPLAN Notices, vol. 19, no. 6; June 1984; 190-201.

[Kent_etal] S. Kent, S. Gaito, and N. Ross; *A meta-model semantics for structural constraints in UML;* Behavioral specifications for businesses and systems, chapter 9, Kluwer Academic Publishers; September 1999; 123-141.

[Kleppe_etal_98] Anneke Kleppe, Jos Warmer, Steve Cook; *Informal formality? The Object Constraint Language and its application in the UML metamodel;* «UML» '98 Beyond the Notation; June 1998; 127-136.

[Knuth_68] D. E. Knuth; *Semantics of context-free languages;* Mathematical Systems Theory, Volume 2; 1968; 127-146.

[Krafft_81] D. Krafft; *A System for the Interactive Development of Verifiably Correct Programs;* Ph.D. dissertation, Dept. Computer Science, Cornell University, Ithaca, N.Y.; August 1981.

[Lano_Bicarregui] K. Lano, J. Bicarregui; *Semantics and Transformations for UML Models;* «UML» '98 Beyond the Notation; June 1998; 97-106.

[Larchevêque_95] J.-M. Larchevêque; *Optimal Incremental Parsing;* ACM Transactions on Programming Languages and Systems, Vol. 17; 1995; 1-15.

[Lemesle_98] R. Lemesle; *Transformation Rules Based on Meta-modeling;* EDOC'98, San Diego; November 1998.

[Levine_etal_92] John Levine, Tony Mason, Doug Brown; *lex & yacc, Second Edition;* O'Reilly UK; ISBN: 1565920007; December 1992.

[Lewin_00] D. Lewin; *Film Strip Editor;* MSc Thesis, University of Kent at Canterbury.; September 2000.

[Linington_99apr] P.F. Linington; *RISCSIM - A Simulator for Object-based Systems;* Proc. UKSIM'99 Conference of the UK Simulation Society; April 1999; 141-147.

[Linington_99sep] Peter F. Linington; *Options for Expressing ODP Enterprise Communities and Their Policies by Using UML;* 3rd International Enterprise Distributed Object Computing Conference (EDOC '99); September 1999.

[Mandel_Cengarle_99] Luis Mandel, María Victoria Cengarle; *On the Expressive Power of OCL;* FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, Springer LNCS 1708; September 1999; 854-874.

[Marlin_96] Chris Marlin; *Multiple views based on unparsing canonical representations--the MultiView architecture;* Joint proceedings of the second international software architecture workshop (ISAW-2)

and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, San Francisco, CA USA; October 1996; 222-226.

[Martin_Utton] G. Martin, P. Utton; *PERMABASE Conceptual Framework;* British Telecommunications plc, 9334:PERMABASE:BT:016; February 1999.

[Medina-Mora_82] R. Medina-Mora; *Syntax-Directed Editing: Towards Integrated Programming Environments;* Ph.D. dissertation, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, Pa.; March 1982.

[Meyers_93] Scott Douglas Meyers; *Representing Software Systems in Multiple-View Development Environments;* Thesis, Department of Computer Science, Brown University, Providence, Rhode Island; May 1993.

[Nuseibeh_etal_94] Bashar Nuseibeh, Jeff Kramer, Anthony Finkelstein; *A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification;* IEEE Transactions on Software Engineering, Volume 20, Issue 10, IEEE Computer Society Press; October 1994; 760-773.

[OMG_00aug] Joint Submission - Alcatel, I-Logix, Kennedy-Carter, Kabira Technologies Inc, Rational Software Corporation, Telelogic AB; *Response to - Action Semantics for UML RFP;* OMG; August 2000.

[OMG_00feb] Jointly by, Data Access Corporation, DSTC, Genesis Development Corporation, Telelogic AB, UBS; *Revised Submission against the UML Profile for CORBA RFP;* OMG Document ad/00-02-02; February 2000.

[OMG_00jan] OMG, Joint submission; *Common Warehouse Metamodel (CWM) Specification;* OMG, Document ad/00-01-01; January 2000.

[OMG_00mar] OMG; *Meta Object Facility (MOF) Specification (version 1.3);* OMG, Document formal/00-04-03; March 2000.

[OMG_00oct] Object Management Group; *The Common Object Request Broker: Architecture and Specification, Revision 2.4;* OMG Document formal/00-10-33; October 2000.

[OMG_01feb] OMG, Architecture Board MDA Drafting Team; *Model Driven Architecture: A Technical Perspective;* OMG, Document ab/2001-02-04; February 2001.

[OMG_98nov] OMG; *Action Semantics for UML RFP;* OMG Document ad/98-11-01; Nov 1998.

[OMG_98oct] Joint Submission:; *XML Metadata Interchange (XMI), Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF);* OMG Document ad/98-10-05; October 1998.

[OMG_99aug] OMG; *UML 2.0 Request For Information;* OMG - ad/99-08-08; August 1999.

[OMG_99dec] Tony Clark, Andy Evens, Robert France, Stuart Kent, Bernard Rumpe; *Response to UML 2.0 Request for Information, Submitted by the precise UML group;* OMG - ad/99-12-16; December 1999.

[OMG_99jun] OMG; *OMG Unified Modeling Language Version 1.3;* OMG, Document ad/99-06-08; June 1999.

[OMG_99jun2] OMG; *C++ Language Mapping Specification;* OMG, Document formal/99-07-41; June 1999.

[OMG_99jun3] OMG; *OMG IDL To Java Language Mapping;* OMG, Document formal/99-07-53; June 1999.

[Övergaard] Gunnar Övergaard; *A Formal Approach to Relationships in The Unified Modeling Language;* Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques; 1998.

[Pane_Myers_96] Pane, J. F. and B. A. Myers; *Usability Issues in the Design of Novice Programming Systems;* Technical Report CMU-CS-96-132, Pittsburgh, PA, Carnegie Mellon University; 1996.

[Peltier_etal_00] Mikaël Peltier, François Ziserman, Jean Bézivin; *On levels of model transformation;* XML Europe 2000, Paris, France; June 2000.

[Pratt_71] T.W. Pratt; *Pair Grammars, Graph Languages and String-to-Graph Translations;* Journal of Computer and System Sciences, Volume 5, San Diego: Academic Press; 1971; 560-595.

[pUML] *The precise UML group;* http://www.cs.york.ac.uk/puml/.

[Rational] *Rational Rose;* Rational Software Corporation, http://www.rational.com.

[Reiss_85mar] Steven P. Reiss; *PECAN: Program Development Systems that Support Multiple Views;* IEEE Transactions on Software Engineering, Volume 11, Issue 3; March 1985; 276-285.

[Reiss_90jun] Steven P. Reiss; *Interacting with the Field environment;* Software-Practice and Experience, Volume 20(S1); June 1990; S1/89 - S1/115.

[Rekers_94] J. Rekers; *On the use of Graph Grammars for defining the Syntax of Graphical Languages;* Proceedings of the colloquium on Graph Transformation, Palma de Mallorca (Technical Report, 94-11, Leiden University, The Netherlands); 1994.

[Rekers_Schürr_96] J. Rekers, A. Schürr; *A Graph Based Framework for the Implementation of Visual Environments;* Proc. VL'96 12th Int. IEEE Symp. on Visual Languages, Los Alamitos: IEEE Computer Society Press; September 1996; 148-155.

[Rekers_Schürr_97] J. Rekers, A. Schürr; *Defining and Parsing Visual Languages with Layered Graph Grammars;* Journal of Visual Languages and Computing, Vol. 8, No. 1, London: Academic Press; 1997; 27-55.

[Reps_etal_83] T. Reps, T. Teitelbaum, A. Demers; *Incremental Context-Dependent Analysis for Language-Based Editors;* ACM Transactions on Programming Languages and Systems, Volume 5, No. 3; July 1983; 449 - 477.

[Reps_Teitelbaum_84] T. Reps, T. Teitelbaum; *The Synthesizer Generator;* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM; 1984; 42-48.

[Reps_Teitelbaum_88] T. Reps, T. Teitelbaum; *The Synthesizer Generator: A System for Constructing Language-Based Editors;* Springer-Verlag, NY; 1988..

[Rozenberg_97] edited by Grzegorz Rozenberg; *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, Foundations;* Singapore, London, World Scientific, ISBN: 9810228848; 1997.

[Rumbaugh_etal_99] James Rumbaugh, Ivar Jacobson, Grady Booch; *The Unified Modeling Language Reference Manual;* Addison Wesley Longman Inc.; 1999.

[Saksena_etal_98] Monika Saksena, Maria M. Larrondo-Petrie, Robert B. France, Mathew P. Evett; *Extending Aggregation Constructs in UML;* «UML» '98 Beyond the Notation; June 1998; 273-280.

[Schürr_94jun] A. Schürr; *Specification of Graph Translators with Triple Graph Grammars;* Proc. WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 903, Berlin: Springer Verlag; June 1994; 151-163.

[Schürr_94nov] A. Schürr; *PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems;* Technical Report AIB 94-11, RWTH Aachen, Germany; 1994.

[Schürr_97] A. Schürr; *Developing Graphical (Software Engineering) Tools with PROGRES, Formal Demonstration;* Proc. 19th Int. Conf. on Software Engineering ICSE'97: IEEE Computer Society Press; May 1997; 618-619.

[Schürr_etal_95] A. Schürr, A. Winter, A. Zündorf; *Graph Grammar Engineering with PROGRES;* Proc. of the 5th European Software Engineering Conference (ESEC'95): LNCS 989, Springer-Verlag; September 1995; 219-234.

[Schürr_Winter_98oct] A. Schürr, A. Winter; *Formal Definition of UML's Package Concept;* The Unified Modeling Language - Technical Aspects and Applications, 1st GROOM UML Workshop Proc.; Oct. 1998; 144-159.

[Shum_Hammond_94] S. Buckingham Shum and N. Hammond; *Argumentation-based design rationale: what use at what cost?;* International Journal of Human-Computer Studies 40 (4); 1994; 603-652.

[Steen_Derrick_00] Maarten Steen, John Derrick; *ODP Enterprise Viewpoint Specification;* Computer Standards and Interfaces; September 2000; 165-189.

[Teitelbaum_Reps_81] Tim Teitelbaum, Thomas Reps; *The Cornell program synthesizer: a syntax-directed programming environment;* Communications of the ACM, Volume 24 , Issue 9, ISSN: 0001-0782; 1981; 563-573.

[Tirri_Lindén_94] Henry Tirri and Greger Lindén; *ALCHEMIST - an object-oriented tool to build transformations between Heterogeneous Data Representations;* Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences (HICSS '94), Maui, Hawaii, volume II, IEEE Computer Society Press; January 1994; 226-235.

[Together] TogetherSoft; *Together Programming Environment (TogetherJ);* http://www.togethersoft.com.

[Utton_Hill] Peter Utton, Brian Hill; *Performance Prediction: an Industry Perspective (Extended Abstract);* Computer Performance Evaluation, Proceedings of the 9th International Conference on Modelling Techniques and Tools (Lecture Notes in Computer Science 1245); June 1997; 1-5.

[Utton_Martin] Peter Utton, Gino Martin; *Further Experiences with Software Performance Modelling;* Proceedings of the First International Workshop on Software and Performance, WOSP 98; October 1998; 14-15.

[W3C_00aug] W3C, Editor Jon Ferraiolo; *Scalable Vector Graphics (SFG) 1.0 Specification, W3C Candidate Recommendation;* W3C; August 2000.

[W3C_98feb] Editors: Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler; *Extensible Markup Language (XML) Version 1.0;* W3C Recommendation, REC-xml-19980210; February 1998.

[W3C_99nov] Editor: James Clark; *XSL Transformations (XSLT) Version 1.0;* W3C Recommendation, REC-xslt-19991116; November 1999.

[Wagner_Graham_98] Tim A. Wagner and Susan L. Graham; *Efficient and flexible incremental parsing;* ACM Transactions on Programming Languages and Systems, Volume 20 , Issue 5; 1998; 980-1013.

[Warmer_Kleppe] Jos B. Warmer, Anneke G. Kleppe; *The Object Constraint Language : Precise Modeling With Uml;* Addison Wesley Publishing Company; October 1998.

[Waters_etal] Gill Waters, Peter Linington, David Akehurst; *Permabase: Predicting the performance of distributed systems at the design stage;* IEE Proceedings - Software; to appear.

[Wegman_80] M.N. Wegman; *Parsing for structural editors;* Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science, IEEE Press, New York; 1980; 320 327.

[Wilson_72] Robin J. Wilson; *Introduction to Graph Theory;* Longman Group Limited; 1972.

[Yeh_Kastens_88] D. Yeh, U. Kastens; *Automatic construction of incremental LR(1) parsers;* ACM SIGPLAN Not. 23, 3; March 1988; 33-42.