This is to certify that the
thesis entitled

Specifying Compositional Semantic Functions For Non-Hierarchical Languages Using Natural Deduction Systems

presented by

Heather J. Goldsby

has been accepted towards fulfillment
of the requirements for the

M. S. degree in Computer Science

R. E. K. Stuart

Major Professor's Signature

4/28/04

Date

SPECIFYING COMPOSITIONAL SEMANTIC FUNCTIONS FOR NON-HIERARCHICAL LANGUAGES USING NATURAL DEDUCTION SYSTEMS

By

HEATHER J. GOLDSBY

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science

2004

ABSTRACT

SPECIFYING COMPOSITIONAL SEMANTIC FUNCTIONS
FOR NON-HIERARCHICAL LANGUAGES
USING NATURAL DEDUCTION SYSTEMS

By

HEATHER J. GOLDSBY

UML-based integrated development environments (IDEs) enable designers to create and type-check UML designs. Some of these IDEs can generate implementation language code; whereas others translate diagrams into a formal notation that can be analyzed by automatic verification techniques. All of these approaches, however, involve the translation of a UML diagram (or set of related diagrams) into some alternative representation. As UML matures and is used in a wider variety of application domains, we expect the number of such translations to increase. This paper describes a notation for specifying custom UML translations, from which lightweight translators can be generated. Moreover, we provide a set of algorithms organized into a tool for ensuring these specified translations are well-formed.

To my Grammie, Jane Goldsby, who always said that education is something no one can take away from you

Acknowledgements

I wish to thank Dr. Stirewalt for teaching me nearly everything I know about computer science and writing. As I complete this masters and look forward to pursing my doctorate, I realize I have so much more to learn, but I am looking forward to the process. Also, Min Deng and Dr. Betty Cheng whose research is correlated with what is presented here and helped me so much along the way.

Finally, my parents, Douglas and Jill Goldsby, who were there when I needed them them most and supported me through all of my difficult times. I couldn't have made it through without them.

Table of Contents

LIST OF FIGURES vii

1 Introduction 1

2 Background 4
- 2.1 UML and Metamodels . . . 4
  - 2.1.1 Class Diagram Notation . . . 7
  - 2.1.2 Formally Representing a Metamodel . . . 8
- 2.2 Programming Language Theory . . . 13
  - 2.2.1 Semantic Functions . . . 13
  - 2.2.2 Expressions . . . 15
- 2.3 Natural Deduction Systems . . . 18
  - 2.3.1 Inference Rules . . . 18
  - 2.3.2 Derivations . . . 20

3 Compositional Semantics for Non-Hierarchical Languages 22
- 3.1 Problem: Assigning Compositional Semantics to Non-Hierarchical Language Expressions . . . 22
- 3.2 Solution: Hierarchical Abstract Syntax Based On Projections . . . 23
  - 3.2.1 Projection . . . 23
  - 3.2.2 Hierarchical Abstract Syntax . . . 27

4 Specifying Semantic Functions Using NDS 31
- 4.1 Semantic Functions Operating Over Projection Type ASTs . . . 31
- 4.2 Optimization . . . 34
- 4.3 Procedure for generating an LNDS . . . 36
- 4.4 Benefit: Symmetric Premise . . . 37
- 4.5 Formally Representing LNDS . . . 39

5 LNDS Example 44
- 5.1 Type layer . . . 44
- 5.2 Impl layer . . . 46
- 5.3 Example derivation . . . 48

6 Validation 52
- 6.1 Automatically checking if an LNDS is well-formed . . . 52
  - 6.1.1 Adding Marked Associations . . . 54
  - 6.1.2 Collapsing Generalizations . . . 56
  - 6.1.3 Eliminating Reflexive Aggregations . . . 57
  - 6.1.4 Checking if a Graph is Cyclic . . . 58

7 Related Work 60

8 Conclusion 62

LIST OF REFERENCES **64**

List of Figures

2.1 UML metamodel 6
2.2 Expression Data Model Example 17
2.3 Sample NDS 19
2.4 Sample Derivation Tree 20

3.1 Projection of an Expression Data Model Example 25
3.2 Projection Types 27
3.3 small class diagram 29
3.4 ASG to AST conversion 29

4.1 Example NDS Built on HAS 32
4.2 Example Derivation applying NDS 33
4.3 C++ AST 33
4.4 Example LNDS 35
4.5 Example Derivation applying LNDS 37
4.6 Example of Symmetric Premises 38

5.1 LNDS Example (1) 45
5.2 LNDS Example (2) 46
5.3 LNDS Example (3) 47
5.4 LNDS Derivation Tree Example 49
5.5 UML ASG 49
5.6 Constructed C++ AST 50
5.7 Code resulting from application of rule Class_{impl} 51

6.1 HIJ language 53
6.2 HIJ LNDS 53
6.3 HIJ language after adding marked associations 56
6.4 HIJ language after collapsing generalizations 57
6.5 HIJ language after removing reflexive aggregations 58
6.6 HIJ graph to be checked for cycles 59

Chapter 1: Introduction

UML-based integrated development environments (IDEs) enable designers to create and type-check UML designs. Some IDEs can generate code from a UML model. For example Rational Rose [31] and Poseidon [20] can generate stubbed class definitions from class diagrams, and some research tools can translate state diagrams into formal specifications that can be analyzed using automatic verification techniques [27, 25, 21]. In addition to fully automated generation, many object-oriented design methods include heuristics for refining UML class diagrams into code or database artifacts [16, 5]. All of these approaches involve the translation of a UML diagram (or set of related diagrams) into some alternative representation. As UML matures and is used in a wider variety of application domains, we expect the number of such translations to increase. Each translation will need to be implemented by a *translator*, which will need to be programmed and integrated into an IDE. This thesis describes a notation for specifying custom UML translations, from which translators can be generated.

This research builds upon the results of the Amalia project, which addresses how to integrate formal analysis capability into IDEs [36, 12, 37, 13]. Using the Amalia tool suite, a designer declaratively specifies an analysis with axioms and inference rules organized in a *natural deduction system* (NDS) [14]. Morevoer, the Amalia generator produces analyzers, which compute behavioral analysis as a side effect of traversing the internal representation of specifications in a formal language (e.g., LOTOS and linear temporal logic), from a NDS. Unfortunately, it is not clear how to use NDS to specify UML translations, thus we cannot directly use the Amalia tool suite to generate UML translators.

Intuitively, a translation is a *semantic function*, i.e., a function that maps programs or models, such as UML diagrams, to target artifacts, such as C++ class stubs,

Promela specifications, or relational database schemas. Traditionally, semantic functions are designed *compositionally*, which means that the meaning of a program is defined in terms of the meaning of its subprograms. Compositional semantic specifications are designed to assume that the syntax of the source language is *hierarchical*, by which we mean programs in the language can be represented as tree structures [34]. UML's syntax is not hierarchical [15]; hence its diagrams are represented not as trees, but rather as labeled graphs. Thus, it is not obvious how to define a semantic function whose domain consists of UML diagrams.

To define a semantic function that operates over UML diagrams, we transform the labeled graphs to trees and then use traditional techniques to define a semantic function that operates over these trees. Thus, we can think of a semantic function whose domain consists of labeled graphs as the composition of two functions: 1) a function that transforms the labeled graphs to trees; 2) a semantic function whose domain consists of those trees. The primary contribution of this thesis is a method for writing these semantic functions in such a way that the intermediary step of constructing the trees is implicit.

To accomplish this, we introduce the notion of a *projection*, which is a collection of types that provide a limited view of the objects in a labeled graph. Projections form trees, which are elaborated in a demand driven fashion by incrementally traversing the graph structure being projected. Thus, the implementation of a semantic function over projections will, as a side effect of traversing (and therefore elaborating) a projection tree, traverse the nodes and edges of the labeled graph. In fact, we show how such a function can be implemented without explicitly instantiating the projection trees.

This thesis contributes: (1) a method for using NDS to write semantic functions that assign meaning to labeled graphs without explicitly converting the graphs to trees, and (2) a tool that checks whether these specifications are well-formed. The re-

mainder of this paper is organized as follows. We first introduce the key background components. Specifically, we introduce the UML notation, programming language theory, and NDS (**Chapter 2**). We then formally define a method for projecting labeled graphs. (**Chapter 3**). Next, we define a method for writing semantic functions that operate over these projections. We then introduce a syntactic extension to NDS that allows us to write semantic functions that operate over labeled graphs. (**Chapter 4**). Specifications written using our syntactic extension of NDS have the potential to be ill-formed, in which case the translator may traverse source graphs without terminating. We provide algorithms for checking that semantic functions written using our extensions are well-formed (**Chapter 6**) and conclude with a discussion of lessons learned and future work (**Chapter 8**).

Chapter 2: Background

Our overall goal is to write semantic functions from which we can automatically generate UML translators. This thesis contributes a method for compositionally assigning meaning to non-hierarchical language diagrams, represented as labeled graphs, and a method for implementing such specifications using graph traversal algorithms. We rely upon the contributions of [11] for extending the syntax of the target language so as to simplify semantic specification. In addition, our work builds upon the results and terminology of three primary fields: UML modeling (**Chapter 2.1**), programming language theory (**Chapter 2.2**), and NDS (**Chapter 2.3**). We build upon the terms and techniques of programming language theory, which is concerned with specifying and implementing semantic functions that operate over trees, to specify and implement semantic functions that operate over labeled graphs. In the sequel, we present algorithms, which analyze if such a semantic specification is well formed, by appealing to a formalization of the metamodel of the source language. We present this formalization in **Chapter 2.1.2**.

2.1 UML and Metamodels

The *United Modeling Language* (UML) [7] is a collection of twelve diagram languages used for specifying and documenting the artifacts of an object-oriented software system. We are interested in translating UML *class diagrams*, which illustrate the static structure of a system in terms of classes and relationships between classes. In addition to being the source language of our translations, we also use UML to represent the syntax of the various languages involved in translation (including UML itself) using *metamodels*, which are class diagrams that model the syntax of a language. For example, **Figure 2.1** depicts a metamodel of the UML class diagram

langauge, which we adapted from [5]. We now introduce the class diagram notation and a formal representation of the metamodel in Z [35].

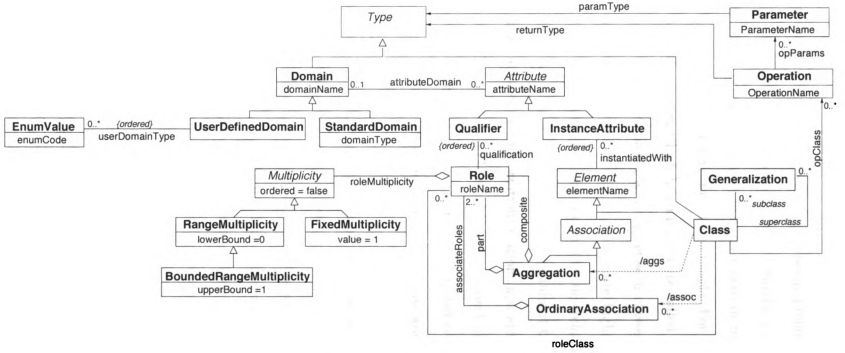


Figure 2.1: UML metamodel

2.1.1 Class Diagram Notation

A UML class diagram depicts a collection of classes and relationships among these classes. A *class* is used to denote a set of objects with similar properties. Graphically a class is depicted as a rectangle. Classes may have zero or more *attributes*, which are listed below the name of the class within the rectangle. For example, **Figure 2.1** depicts a class named Parameter, which has an attribute called ParameterName.

Relationships in UML are depicted as lines between classes. The *generalization* relationship is used to express that one class is a more highly specialized version of another class. The most general class is called the *superclass*, whereas the more specialized classes are called the *subclasses*. Subclasses inherit the attributes, and relationships of their superclass. Generalization relationships are graphically depicted by a triangle attached to the superclass with lines connecting the triangle to the subclasses. An example generalization is the relationship between Type, Domain and Class, in which Type is the superclass and Domain and Class are the subclasses (**Figure 2.1**).

The *association* relationship is used to express an interaction between instances of classes. An association is depicted as a line that connects two classes[1], whose instances are said to *play a role* in the association. Generally, associations are directed and hence are depicted as arrows. An association depicted as a line without an arrow is a *bidirectional association.* Associations are named, and each end of an association can be given a name to further clarify the role that objects of the adjacent class play in the association. Each role has a *multiplicity* that indicates the number of instances of the adjacent class that will be linked with a single instance of the peer class under the association. An example of a bidirectional association is roleClass, which relates instances of class Class and class Role (**Figure 2.1**).

[1]It is possible to have an association that relates a class to itself. Such an association is called *reflexive.*

An *aggregation* is a special form of association that denotes one class is a part of another class. Aggregations are depicted as a line connecting two different classes with a diamond on the line adjacent to the container class. For example, the *composite* association depicted in **Figure 2.1** is an aggregation that relates instances of class **Aggregation** to instances of class **Role** (**Figure 2.1**).

2.1.2 Formally Representing a Metamodel

In the sequel, our algorithms determine if a semantic function specified in a variant of NDS is well-formed by appealing to the metamodel of the source language. Hence, we need to represent metamodels as data types[2]. We have chosen to write the specification of the metamodel data type in Z [35] to conform to related research projects, which are also formalizing the UML metamodel using Z [11].

Fundamentally, a metamodel is a set of classes, a set of associations and a generalization relation. To model these concepts, we introduce the following given sets.

$$[CHAR]$$

$$STRING == \text{seq}\ CHAR$$

The *CHAR* set is used to form strings from which we can represent the names of classes, attributes, and associations.

$$[CLASS\_ID, ATTR\_ID, ASSOC\_ID]$$

CLASS_ID, *ATTR_ID*, *and ASSOC_ID* comprise the identifiers for classes, attributes and associations respectively. Elements of these sets can only be compared for equality; we use them to define accessor functions, which, for instance, retrieve a

[2]Our formalization models only the information needed by our algorithms, not all the information contained within metamodels.

class's attributes given a *CLASS_ID*.

Each class in **Figure 2.1** corresponds to some element in the set *CLASS_ID*. When referring to a specific *CLASS_ID*, we use the name of the corresponding class rendered in Helvetica font. For example, Parameter refers to the *CLASS_ID* that corresponds to the class named "Parameter" in **Figure 2.1**. We adopt a similar convention for elements of sets *ASSOC_ID* and *ATTR_ID*. These conventions assume that association and attribute names are unique in a model[3].

Modeling the meaning of an attribute requires a space of legal data types, which we model with the following free type:

$$DATATYPE ::= boolean \mid integer \mid string$$

In this formalization, the type of an attribute can be either boolean (boolean), integer (integer), or string (string).

Each class attribute comprises a name, and a data type. We formalize this notion with a tuple:

$$ATTRIBUTE == STRING \times DATATYPE$$

For instance, the attribute ParameterName (**Figure 2.1**) is represented as the tuple ("ParameterName", string).

The accessor function *getAttribute* retrieves the attribute tuple that corresponds to an attribute identifier. It is formally defined as follows.

$$getAttribute : ATTR\_ID \rightarrow ATTRIBUTE$$

Applying the *getAttribute* function to the attribute identifier ParameterName yields the tuple ("ParameterName", string).

[3]This assumption is not valid for general UML class models, but it greatly simplifies our treatment; thus we impose the restriction on **Figure 2.1**.

Formally, each class comprises a name and a set of attribute identifiers. We represent this in Z as follows.

$$CLASS == STRING \times \mathbb{P}\, ATTR\_ID$$

For instance, the class Parameter (**Figure 2.1**) is represented as the tuple ("Parameter", {ParameterName}). The accessor function *getClass* returns the class tuple that corresponds to a given *CLASS_ID*. The accessor functions *getClassName* and *getClassAttrs* return a string that is the class's name and the the set of attribute identifiers that comprise a class respectively. These functions are formalized as follows.

$$\begin{array}{|l} getClass : CLASS\_ID \rightarrow CLASS \\ getClassName : CLASS \rightarrow STRING \\ getClassAttrs : CLASS \rightarrow \mathbb{P}\, ATTR\_ID \\ \hline getClassName = first[STRING, \mathbb{P}\, ATTR\_ID] \\ getClassAttrs = second[STRING, \mathbb{P}\, ATTR\_ID] \\ \forall\, c1, c2 : CLASS\_ID;\ class1, class2 : CLASS \bullet \\ \quad getClassAttrs(class1) \cap getClassAttrs(class2) \neq \varnothing \Leftrightarrow class1 = class2 \end{array}$$

For example, applying the function *getClass* to Parameter yields the tuple ("Parameter", { ParameterName }). Applying the function *getClassName* to class represented by the tuple ("Parameter" {ParameterName}) yields "Parameter" and applying *getClassAttrs* to the same tuple yields {ParameterName}.

We also define relation *genRelation* to represent generalization relationships in the metamodel. Specifically, the *genRelation* relates two class identifiers. The first class identifier is that of the superclass and the second class identifier is that of the subclass. Formally, this relation is specified as follows.

$$\begin{array}{|l} genRelation : CLASS\_ID \leftrightarrow CLASS\_ID \\ \hline \forall\, c : CLASS\_ID \bullet \\ \quad (c, c) \notin genRelation^{+} \end{array}$$

The invariant states that there are no cycles in the generalization hierarchy. If c is a *CLASS_ID*, then applying $genRelation(\!|\ \{c\}\ |\!)$ returns the set of subclasses of c. For instance, $genRelation(\!|\ \{\mathsf{Type}\}\ |\!)$ yields the set $\{\mathsf{Domain}, \mathsf{Class}\}$.

Classes relate to one another by associations. There are multiple types of associations, which we define by the free type:

$$\begin{aligned} ASSOCIATION\_TYPE ::= & \ aggregation\langle\langle CLASS\_ID \times CLASS\_ID \rangle\rangle \\ | & \ directed\langle\langle CLASS\_ID \times CLASS\_ID \rangle\rangle \end{aligned}$$

An association can be either an aggregation with a composite class and part class, or a directed association with a source class and a target class. We model a bidirectional association as two directed associations. The bidirectional association roleClass (**Figure 2.1**) is represented as directed (Class, Role) and directed (Role, Class). Likewise, the aggregation part (**Figure 2.1**) is represented as aggregation (Aggregation, Role).

We define several helper functions to return pertinent information about *ASSOCIATION_TYPE*s. They are as follows:

$$\begin{array}{|l} getAssocSrcClass : ASSOCIATION\_TYPE \rightarrow CLASS\_ID \\ getAssocTargClass : ASSOCIATION\_TYPE \rightarrow CLASS\_ID \\ \hline \forall\, c1, c2 : CLASS\_ID \bullet \\ \quad getAssocSrcClass\ aggregation(c1, c2) = c1 \\ \quad \wedge \\ \quad getAssocSrcClass\ directed(c1, c2) = c1 \\ \quad \wedge \\ \quad getAssocTargClass\ aggregation(c1, c2) = c2 \\ \quad \wedge \\ \quad getAssocTargClass\ directed(c1, c2) = c2 \end{array}$$

The functions *getAssocSrcClass* and *getAssocTargClass* when applied to an *ASSOCIATION_TYPE* element return the first and second class identifier respectively. For example, applying the *getAssocSrcClass* function to the part aggregation (**Figure 2.1**), represented by the element aggregation (Aggregation, Role), yields Ag-

gregation. Applying the *getAssocTargClass* function to the same element yields **Role.**

An association comprises a name and a type as follows:

$$ASSOCIATION == STRING \times ASSOCIATION\_TYPE$$

For example, the **part** aggregation is represented as the tuple (**"part", aggregation (Aggregation, Role)**).

We define a function that returns the tuple representing an association when given an association identifier. We also define accessor functions to return the first and second elements of a tuple representing an association. They are formalized as follows.

$$\begin{array}{|l} getAssoc : ASSOC\_ID \rightarrow ASSOCIATION \\ getAssocName : ASSOCIATION \rightarrow STRING \\ getAssocType : ASSOCIATION \rightarrow ASSOCIATION\_TYPE \\ \hline \forall\, a : STRING;\ at : ASSOCIATION\_TYPE \bullet \\ \quad getAssocName = first[STRING, ASSOCIATION\_TYPE] \\ \quad getAssocType = second[STRING, ASSOCIATION\_TYPE] \end{array}$$

For example, applying *getAssoc* to *ASSOC_ID* (**part**) yields (**"part", (aggregation, Aggregation, Role)**).

We model a metamodel as a set of classes and a set of associations. It is formalized as follows.

Metamodel

$$\begin{array}{|l} classes : \mathbb{P}\, CLASS\_ID \\ assocs : \mathbb{P}\, ASSOC\_ID \\ \hline classes = \mathrm{dom}\, getClass \\ assocs = \mathrm{dom}\, getAssoc \end{array}$$

2.2 Programming Language Theory

Our work is concerned with defining semantic functions, hence we build upon the terminology of programming language theory. However, because this thesis is concerned with specifying semantic functions that operate over labeled graphs, which is not addressed by programming language theory, we also introduce terms we created.

2.2.1 Semantic Functions

The programming-language semantics community uses NDS as a formal method for specifying semantics [34]. Moreover, the Amalia framework is capable of generating analyzers from semantic specifications written using NDS [36]. Thus, we use NDS to formally specify semantic functions that translate UML to an implementation language.

Formally, a semantic function maps elements of the *syntactic domain* to elements of the *semantic domain.* A syntactic domain is the set of trees (or labeled graphs) that represent valid programs (or diagrams) in the source language. A semantic domain is the set of mathematical objects used to give meaning to elements of the syntactic domain of a language [34]. In traditional denotational semantics, the syntactic domain is assumed to be an *algebraic data type*, which is a disjoint union of types that are possibly recursive. Instances (or *terms*) of algebraic data types are constructed from a finite set of *operators*, which are injective functions that map elements of the constituent type into the union. At least one of the operators in an algebraic data type must be nullary. For example, we can define an algebraic data type *Nat* whose elements are constructed using two operators:

$$Zero :\rightarrow Nat$$
$$Succ : Nat \rightarrow Nat$$

where *Zero* is a nullary operator and *Succ* is a unary operator. Terms in this data type

include *Zero*, *Succ*(*Zero*), *Succ*(*Succ*(*Zero*)), etc. A language with hierarchical syntax can trivially be represented as algebraic data type, where the operators correspond to the language syntax constructs and syntactically correct programs in the language can be represented as terms.

A semantic function is defined compositionally by defining cases based on the major operator of the term to which the function is applied. If the major operator is a nullary constructor, then the function is computed based on the basic attributes of the term. By contrast, if the major operator is a constructor with arity > 0, then the function is computed in terms of these basic attributes and the results attained by applying the function recursively to the term's operands. For example, we can define a semantic function $\mathcal{W} : \mathit{Nat} \rightarrow \mathbb{N}$ as follows:

$$\mathcal{W}[\![\text{Zero}]\!] = 0$$
$$\mathcal{W}[\![\text{Succ(x)}]\!] = \mathcal{W}[\![\text{x}]\!] + 1$$

Applying semantic function $\mathcal{W}$ to term *Zero* yields 0. Thus, the pair $(\mathit{Zero}, 0)$ is said to be an element of $\mathcal{W}$. In the sequel, we refer to such pairs as *mapping assertions* [13] and represent them $\mathit{Zero} \mapsto 0$ to allow easy integration into the rule format. Compositional definitions are useful because they encapsulate (in the cases) the meaning of each feature of a language in isolation. Notice, however, that this benefit is achievable because the syntactic domain is an algebraic data type (and thus its terms are hierarchical).

By contrast, programs and diagrams in a non-hierarchical language (such as UML) are represented as labeled graphs, rather than algebraic terms. Labeled graphs lack the hierarchical operator–operand structure of algebraic terms. Consequently, it is not obvious how to define a compositional semantics for such languages.

In the sequel, we show how to transform labeled graphs into algebraic terms, for which a compositional semantics can then be defined. The functional composition of

the labeled graph to algebraic term translation with the semantic function yields a compositional semantic function for translating UML diagrams. We ultimately want to integrate a UML translator into an existing UML modeling tool. Thus, we define the translation from labeled graphs into algebraic terms such that the translation can be implemented to be performed incrementally, thereby allowing the generated UML translator to generate target code as a side effect of traversing the in-memory UML representations (which are labeled graphs). To make these ideas more precise, we now describe how algebraic terms and labeled graphs are implemented in tools as linked object structures.

2.2.2 Expressions

We refer to the implementations of both algebraic data types and labeled graphs as *expressions*. A language's *abstract syntax* refers to a class of expressions that record the structure of programs or specifications in that language. Expressions are generally implementations of algebraic data types and hence are trees; thus we shall use the conventional term *abstract syntax tree* (AST) [32] to refer to expressions representing programs or specifications or to expressions representing (syntactically well-formed) parts of a program or a specification. The implementation of labeled graphs e.g., UML diagrams, are graphs; thus we introduce the term *abstract syntax graph* (ASG) to refer to expressions representing models or to expressions representing (syntactically well-formed) parts of models. All expressions are instantiations of *expression types*, which we formally define as follows.

A *data model* $\mathcal{D}$ is a finite collection of classes, such that for each class $c \in \mathcal{D}$:

- The public interface of c contains no mutable operations, and
- For every class c' that appears in the signature of an operation in the interface of c: $c' \in \mathcal{D}$.

We define the relation $returns_{\mathcal{D}}$: $\mathcal{D} \leftrightarrow \mathcal{D}$, such that $\forall\, c, c' \in \mathcal{D} \bullet (c, c') \in returns_{\mathcal{D}}$ if and only if there exists an operation *op* in the interface of c such that c' is referenced in the return type of *op*.

An *expression data model* $\mathcal{E}$ is a data model with the additional constraint that every parameterized operation[4] of every class $c \in \mathcal{E}$ takes parameters that are values of primitive domains (i.e., `integer`, `string`, etc.) and the return type is either `void`, a value of a primitive domain, or a reference to an instance of a class in $\mathcal{E}$. We refer to the classes comprising an expression data model as expression types and we refer to operations that return references to expression types as *navigation operations.*

Expression types are not necessarily algebraic data types because their instances may be graphs rather than trees.

For example, **Figure 2.2** depicts an expression data model with two expression types, `Class` and `Role`. In this example, `Class` contains one navigation operation `getRoleClass`, and `Role` contains one navigation operation, also called `getRoleClass`. Observe, instances of these expression types form graphs, rather than trees. Thus, this expression data model is not an algebraic data type.

A metamodel is a graphical depiction of an expression data model. Each class in the metamodel corresponds to an expression type. The outgoing associations of a given class in the metamodel are represented as the pertinent expression type's navigation operations. Specifically, an association with name X is represented as a navigation operation named *getX* (i.e., the association name prepended with the keyword *get*). Moreover, *getX* returns a pointer to instances of the expression type, which corresponds to the class at the target end of association X. The navigation operation is a member of the expression type corresponding to the source class of the association. Observe expression types `Class` and `Role` (**Figure 2.2**) correspond to classes Class and Role (**Figure 2.1**). Navigation operations `getRoleClass` corre-

[4]excluding the constructor

```
class Class {
   public:
      Class( const string & name, const Role* role ) :
                _name(name); _role(role) {}

      const string & getName() const { return _name; }
      const Role* getRoleClass() const { return _role; }

   protected:
      const string _name;
      const Role* _role;
   };
```

Expression Type Class

```
class Role {
   public:
      Role( const string & name, const Class* class ) :
                _name(name); _class(class) {}

      const string & getName() const { return _name; }
      const Class* getRoleClass() const { return _class; }

   protected:
      const string _name;
      const Class* _class;
};
```

Expression Type Role

Figure 2.2: Expression Data Model Example

sponds to association roleClass.

2.3 Natural Deduction Systems

The programming-language semantics community use NDS to specify semantic functions that assign meaning to algebraic terms [34]. NDS is able to specify only those semantic functions whose syntactic domain is an algebraic data type. The implementation of these semantic functions operate over ASTs, the in-memory representation of algebraic terms.

2.3.1 Inference Rules

An *inference rule* prescribes how to derive mapping assertions for an algebraic term represented as a tree from the mapping assertion derivable from its subtrees. An inference rule takes the following form:

$$\frac{\begin{array}{lcl} c_1 & \mapsto & d_1 \\ & \vdots & \\ c_n & \mapsto & d_n \end{array}}{A(c_1, ..., c_n) \mapsto B(d_1, ..., d_n)} \text{ [rulename]}$$

Informally, the rule states that if the subtrees $c_1, ..., c_n$ translate into $d_1, ..., d_n$ then the tree $A(c_1, ..., c_n)$ translates into $B(d_1, ..., d_n)$. Mapping assertions in the *numerator* of a rule ($c_1 \mapsto d_1$, ..., $c_n \mapsto d_m$) make up the rule's *premises*. A rule without premises is called an *axiom*. The *denominator* of a rule ($A(c_1, ..., c_n) \mapsto B(d_1, ..., d_n)$) specifies a mapping assertion, called the *conclusion*, that is inferred using the rule.

Each side of the conclusion is called an *AST schema* because traditionally the implementation of each semantic the function will operate over ASTs and construct ASTs. Each AST schema comprises an *operator* (e.g., A, B), which is an expression

$$\frac{}{Zero \;\mapsto\; 0}\ [\text{Zero}]$$

$$\frac{x \;\mapsto\; y}{Succ(x) \;\mapsto\; PlusOne(y)}\ [\text{Succ}]$$

Figure 2.3: Sample NDS

type, and a series of parameters called *free variables* (e.g., $(c_1, ..., c_n)$ and $(d_1, ..., d_n)$) . These free variables will be bound to either strings or instantiated AST schemas called *operands.*

Figure 2.3 depicts a sample NDS that specifies the semantic function $\mathcal{W}$ referenced earlier. Specifically, the AST schemas for axiom [*Zero*] comprise operators *Zero* and 0; neither has free variables. The source AST schema for inference rule [*Succ*] comprises an operator, *Succ*, and a free variable, x, which is a place holder for an operand. Similarly, the target AST schema for inference rule [*Succ*] comprises an operator, *PlusOne*, and a free variable, y, which is a place holder for an operand. The single premise of this inference rule maps free variable x to free variable y.

We adopt several conventions regarding the naming of operators and free variables. Because the subject operator is an expression type of the language to be translated, it will be depicted as a class in the metamodel depiction of the language's abstract syntax. We name free variables that parameterize the subject after the constructs in the metamodel. Free variables used as place holders for operands are named after the association that relates the class representing the subject operator to the class representing the expression type of the operand. For example, the metamodel for the natural numbers language would contain classes named *Succ*, *Zero*, and *Nat*, and a directed association or aggregation named x connecting *Succ* to *Nat*. Moreover, *Succ* and *Zero* would inherit from the abstract class *Nat*. Non-operand free variable placeholders are named after attributes of the classes depicted in the metamodel.

$$\dfrac{\dfrac{\dfrac{}{Zero \mapsto 0}\,[\text{Zero}]}{Succ(Zero) \mapsto PlusOne(0)}\,[\text{Succ}]}{Succ(Succ(Zero)) \mapsto PlusOne(PlusOne(0))}\,[\text{Succ}]$$

Figure 2.4: Sample Derivation Tree

2.3.2 Derivations

A *derivation* is the process of translating a tree in the conclusion of the rule by inductively applying inference rules to translate the sub-operands of the tree until only axioms can be applied. At this point, the original tree can be translated. If NDS specifies a semantic function every pair in the function, has a corresponding derivation tree whose conclusion is the mapping assertion representing the function. This derivation tree proves the conclusion.

Derivation trees are able to prove translation by relying upon a technique called *structural induction*, which is a form of mathematical deduction designed for proving properties by decomposing tree structures. The leaves of a derivation tree correspond to applications of axioms, or applications of the inductive base cases. Each non-leaf node is an application of an inference rule and each edge connects an instantiated premise with the derivation tree that proves its validity, or in our case proves it translates correctly. These are applications of the inductive cases.

For example, **Figure 2.4** depicts the derivation tree that proves $(Succ(Succ(Zero())), PlusOne(PlusOne(0())))$ is an element of the semantic function $\mathcal{W}$. The mapping assertion representing this pair $Succ(Succ(Zero())) \mapsto PlusOne(PlusOne(0()))$ is the root of the derivation tree and is depicted at the bottom of the diagram. This mapping assertion corresponds to the conclusion of an application of rule $[Succ]$. The application of the premise of the rule $[Succ]$ now corresponds to the derivation tree that proves mapping assertion $Succ(Zero()) \mapsto PlusOne(0())$. Thus, the premise has been replaced by another application of rule

[*Succ*], now with conclusion $Succ(Zero()) \mapsto PlusOne(0()))$. The premise of this rule has been replaced with an application of axiom [*Zero*], which proves the mapping assertion $Zero() \mapsto 0()$.

Chapter 3: Compositional Semantics for Non-Hierarchical Languages

Because semantic functions are compositional, writing them for a hierarchical language, whose instances are algebraic terms, is straightforward: The meaning of a composite term is defined in terms of the meanings of its parts. However, writing semantic specifications for a non-hierarchical language, e.g., UML, whose instances are labeled graphs, is non-trivial because traversing a link from a labeled graph returns another labeled graph, which may comprise the original labeled graph. Thus, the meaning of a labeled graph cannot be specified compositionally by recursively deconstructing a composite labeled graph into recursively smaller parts.

3.1 Problem: Assigning Compositional Semantics to Non-Hierarchical Language Expressions

To write semantic specifications for a non-hierarchical language, we make the language appear to be hierarchical by imposing a tree structure on its labeled graphs. Conceptually, this is possible because a graph can be unwound into an tree without any loss of information. The traversal begins at some designated node, from which the root of a tree is created. As a graph is traversed by recursively visiting its graph nodes and traversing their outgoing edges, a new tree node is created for every graph node visited by the traversal, and a new tree edge is created for every graph edge traversed by the traversal. If the graph contains a cycle, infinitely many tree nodes will be created to represent a single graph node, thus the tree is of infinite depth.

Our key contribution is a technique for imposing an algebraic term structure onto labeled graphs to make the latter appear to be finite trees. This technique enables compositional specification of functions over labeled graphs.

To avoid confusion, we have carefully distinguished a semantic function, which maps labeled graphs to terms in an algebraic data type, from the implementation of this function, which traverses an ASG to construct an AST. In the sequel, we use the terms labeled graph and ASG interchangeably and the terms algebraic term and AST interchangeably when the distinction is clear from context.

3.2 Solution: Hierarchical Abstract Syntax Based On Projections

To make ASGs appear to be finite ASTs, we developed a new abstract syntax representation called *hierarchical abstract syntax* (HAS). This representation is based on the primitive notion of a *projection*, which implements the decorator pattern [19]. It is formalized as follows.

3.2.1 Projection

We say a data model $\mathcal{P}$ is a projection of an expression data model $\mathcal{E}$ if there exists a surjective function $h_{\mathcal{P}}$: $\mathcal{P} \twoheadrightarrow \mathcal{E}$ such that $\forall\, c \in \mathcal{P}$

1. The implementation of c declares a data member s whose type is a reference to $h_{\mathcal{P}}(c)$. We shall henceforth refer to such an s as the *source reference* of c.

2. For each operation *op* in the interface of c; the return type of *op* is either `void`, a value of a primitive domain, or an instance of some class $c' \in \mathcal{P}$. Moreover:

   - in the first two cases, there must exist an operation op' in the interface of

$h_{\mathcal{P}}(c)$ such that *op* has exactly the same signature as *op'* [1].

- in the last case, let

  R *opName* (*argList*)

  be the signature of *op*. Then the interface of $h_{\mathcal{P}}(c)$ must contain an operation *op'* with signature

  $h_{\mathcal{P}}(R)$ * *opName* (*argList*)

  In other words, if *op*'s return type, *R*, is an instance of some class *c'*, the interface of $h_{\mathcal{P}}(c)$ must declare an operation *op'* whose *opName* and *argList* are identical to that of *op*, but whose return value is a reference to $h_{\mathcal{P}}(c')$.

If such an $h_{\mathcal{P}}$ exists, we refer to classes in $dom(h_{\mathcal{P}})$ as *projection types* and their corresponding classes as *source expression types.*

For example, **Figure 3.1** depicts a projection of the expression data model depicted in **Figure 2.2**. It has three projection types `Class0`, `Class1` and `Role0`. In this example, `Class` is the source expression type of both `Class0` and `Class1`; `Role` is the source expression type of `Role0`. The interface of `Class0` declares operations `getName` and `getRoleClass`, where `getName` has exactly the same signature as the `getName` operation declared in the interface of `Class`. The signature of `Class0`'s operation `getRoleClass` is

```
Role0    getRoleClass ( )  const
```

[1] An operation's *signature* consists of its name, return type and parameters.

```
class Class0 {
    public:
      Class0( const Class* c ) :  _c(c) {}

      const string& getName() const { return _c->getName(); }
      Role0 getRoleClass() const { return Role0(_c->getRoleClass(); }

   protected:
      const Class* _c;
};
```

Projection Type Class0

```
class Class1 {
   public:
      Class1( const Class* c ) :  _c(c) {}

      const string& getName() const { return _c->getName(); }

   protected:
      const Class* _c;
};
```

Projection Type Class1

```
class Role0 {
   public:
      Role0( const Role* r ) :  _r(r) {}

      const string & getName() const { return _r->getName(); }
      Class1 getRoleClass() const { return Class1( _r->getRoleClass()); }

   protected:
      const Role* _r;
};
```

Projection Type Role0

Figure 3.1: Projection of an Expression Data Model Example

We thus expect the interface of $h_{\mathcal{P}}(Class0)$ to declare an operation with the signature

$h_{\mathcal{P}}$(Role0) * getRoleClass ( ) const

and indeed, the interface of *Class* (**Figure 2.2**) declares such an operation. Similarly, the **getName** operations declared in the interface of **Role0** and **Class1** have exactly the same signature as the **getName** operations declared in the interfaces of **Role** and **Class** respectively (**Figure 2.2**). The signature of the operation **getRoleClass** declared in the interface of **Role0** is

```
Class0    getRoleClass ( )   const
```

We thus expect the interface of $h_{\mathcal{P}}(Role0)$ to declare an operation with the signature

$h_{\mathcal{P}}$(Class0) * getRoleClass ( ) const

and indeed the interface of **Role** (**Figure 2.2**) declares such an operation.

We use projections to unwind labeled graphs into trees. Observe, the types within an expression data model can be instantiated to form an ASG. The simplest case of this occurring is when an expression data type contains a navigation operation that returns a pointer to an instantiation of another expression data type, which in turn contains a navigation operation that returns a pointer to an instantiation of the original expression data type. **Figure 2.2** depicts this scenario. Because projection type operations return instances of projection types (**Figure 3.1**), rather than pointers to instances of projection types, it is impossible for two projection type instances to be mutually referencing. Thus, projection type instances form projection ASTs, rather than ASGs. We further illustrate this point with an example in the following section.

```
class ErrClass0 {
    public:
      ErrClass0( const Class* c ) :  _c(c) {}

      const string& getName() const { return _c->getName(); }
      ErrRole0 getRoleClass() const
           { return ErrRole0(_c->getRoleClass(); }

   protected:
      const Class* _c;
};
```

Projection Type ErrClass0

```
class ErrRole0 {
   public:
      ErrRole0( const Role* r ) :  _r(r) {}

      const string & getName() const { return _r->getName(); }
      ErrClass0 getRoleClass() const
           { return ErrClass0( _r->getRoleClass()); }

   protected:
      const Role* _r;
};
```

Projection Type ErrRole0

Figure 3.2: Projection Types

3.2.2 Hierarchical Abstract Syntax

Although instances of projection types are guaranteed to form ASTs, they are not guaranteed to form finite ASTs. For example, the `ErrRole0` and `ErrClass0` depicted in **Figure 3.2** are valid projection types, but traversals of their instances are not finite. An instance of `ErrRole0` contains operation `getRoleClass` whose return type is an instance of `ErrClass0`. An instance of `ErrClass0` contains operation `getRoleClass` whose return type is an instance of `ErrRole0`. Traversals of instances of these projection types continue to create new instances, which would be traversed

indefinitely. To ensure the ASTs are finite, we define a language's HAS as a collection of projection types with some additional constraints. HAS is formalized as follows.

Let $Layer_{\mathcal{P}}$ be a partition of the projection $\mathcal{P}$, such that:

$\forall \mathcal{S} \in Layer_{\mathcal{P}} \bullet \forall c, c' \in \mathcal{S} \bullet h_{\mathcal{P}}(c) = h_{\mathcal{P}}(c') \Leftrightarrow c = c'$

For each set $\mathcal{S}$ in the partition $Layer_{\mathcal{P}}$ no two projection types in the set have the same source expression type.

A trivial example of an HAS is where each of the partition's sets contains one projection type. Obviously, it is impossible for a given set to contain multiple projection types that have the same source expression type.

$Layer_{\mathcal{P}}$ is a language's HAS if $Layer_{\mathcal{P}}$ can be indexed by a minimal, partially ordered set $\mathcal{I}$ with a maximal index, such that: $\forall i \in \mathcal{I}$:

1. $\nexists c \in Layer_{\mathcal{P}}(i) \bullet (c, c) \in (Layer_{\mathcal{P}}(i) \lhd returns_{\mathcal{P}})^+$

   The $returns_{\mathcal{P}}$ relation relates a projection type to the projection types that are returned by any of its operations. Taking the transitive closure of the $returns_{\mathcal{P}}$ relation with the range restricted to the projection types in $Layer_{\mathcal{P}}(i)$ relates a projection type to the projection types that are reachable by applying the relation multiple times. Thus, the predicate states that a projection type in an HAS is never reachable from itself.

2. $\forall c \in Layer_{\mathcal{P}}(i), \forall c' \in \mathcal{P} \bullet (c, c') \in returns_{\mathcal{P}} \Leftrightarrow \exists j \bullet j \geq i \bullet c' \in Layer_{\mathcal{P}}(j)$

   If the $returns_{\mathcal{P}}$ relation relates two projection types in different sets within the partition, the projection type in the domain of the $returns_{\mathcal{P}}$ relation is in a set whose index is less than or equal to the index of the set of the projection type in the range of the relation.

We refer to $Layer_{\mathcal{P}}(i)$ as a *layer*.

The projection types in **Figure 3.1** form an HAS for an elided portion of UML's abstract syntax. The HAS is partitioned into two layers, `0` and `1`. Layer `0` has

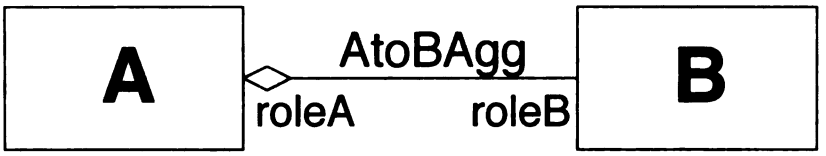
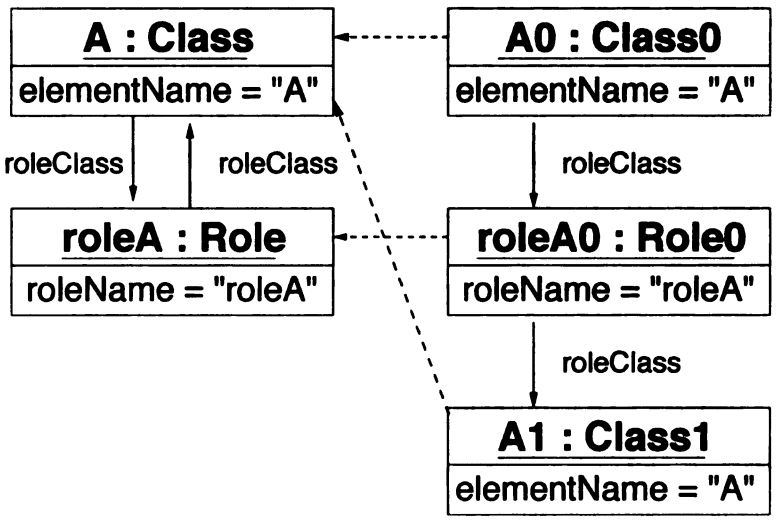


Figure 3.3: small class diagram



a. b.

Figure 3.4: ASG to AST conversion

two elements, `Role0` and `Class0`; whereas layer `1` has one element, `Class1`. To demonstrate the correspondence between a projection type and both its layer and source expression type, we adopt the convention of naming our projection types with the name of their source expression type appended with the name of their layer. `Class0`, `Class1` and `Role0` adhere to this convention.

The HAS we have specified allows us to construct finite ASTs of instances of projection types by traversing labeled graphs. To concretely illustrate this solution, we offer the following example. **Figure 3.3** depicts a UML class diagram with two classes, A and B. Class A is related to class B by aggregation *AtoBAgg*. Class A's role in *AtoBAgg* is *roleA*. Class B's role in *AtoBAgg* is *roleB*. **Figure 3.4 a** depicts a part of the ASG constructed using the UML metamodel depicted in **Figure 2.1**.

Using projection types, traversing the ASG depicted in **Figure 3.4 a** constructs

the projection AST depicted in **b**. **A0** is an instance of `Class0`; **A1** is an instance of `Class1`; and **roleA0** is an instance of `Role0`. **A0** contains a reference to **A**, the instance of its expression type, and a link (called *roleClass*) to **roleA0**. **roleA0** contains a reference to **roleA**, the instance of its expression type, and a link (called *roleClass*) to **A1**. **A1** contains a reference to **A**, the instance of its expression type. The semantic meaning of **A0** depends on the meaning of **roleA0**, which depends on the meaning of **A1**. The meaning of **A1** does not depend on the meanings of any projection type instances. Thus, it is possible to define a compositional semantic function, whose implementation would translate this AST to a target language AST.

Developing an HAS for a non-hierarchical language allows us to transform the language's expressions from ASGs to finite projection ASTs. Because ASTs can be assigned meaning compositionally, we are then able to define a semantic function whose implementation traverses these ASTs and as a by product produces a target language AST, which can be pretty printed to produce implementation language code.

Chapter 4: Specifying Semantic Functions Using NDS

The process we have specified for translating an ASG to a target language AST involves first projecting the ASG to a projection AST and then translating the projection AST to a target language AST by applying the NDS specified semantic function. In this section, we illustrate the second step of this process by specifying a sample semantic function, whose implementation translates projection ASTs to target-language ASTs (**Section 4.1**). We then present a syntactic extension to NDS, which allows us to optimize the process by eliminating the intermediary step of constructing projection ASTs (**Section 4.2**).

4.1 Semantic Functions Operating Over Projection Type ASTs

As an example, we specify an NDS that translates projection ASTs to target language ASTs. This example is selected for its brevity; the rules are not part of the UML to C++ translation we specify later.

Figure 4.1 depicts an NDS that specifies a semantic function whose implementation maps instances of Class1, Class0 and Role0 (**Figure 3.1**) to C++ ASTs. Axiom [*Class*1] translates instances of Class1 to instances of the CppClassForwardDec, which represents the forward declaration of a C++ class. Rule [*Role*0] translates instances of Role0 to instances of FunctionMember, which represents a C++ function member. Rule [*Class*0] translates instances of Class0 to instances of CppClass, which represents a C++ class definition. The parameters of each subject operator may appear as the subject of a premise or as a parameter of the target operator. If a given rule does not

$$\frac{roleClass \mapsto functionMember}{\mathsf{Class0}(elementName, roleClass) \mapsto \mathsf{CppClass}(elementName, functionMember)}\ [Class0]$$

$$\frac{}{\mathsf{Class1}(elementName) \mapsto \mathsf{CppClassForwardDec}(elementName)}\ [Class1]$$

$$\frac{roleClass \mapsto cppClassForwardDec}{\mathsf{Role0}(roleName, roleClass) \mapsto \mathsf{FunctionMember}(roleName, cppClassForwardDec)}\ [Role0]$$

Figure 4.1: Example NDS Built on HAS

reference a subject parameter in its premise or target, then that parameter may be elided. Thus, the subject operator of projection rules with the same expression type, such as [*Class*0] and [*Class*1], have different parameters. Specifically, the subject operator, Class0, of rule [*Class*0], is parameterized by *elementName* and *roleClass*; whereas the subject operator of rule [*Class*1], which is Class1, is parameterized only by *elementName*.

Applying the projection rules shown in **Figure 4.1** to the projection AST depicted in **Figure 3.4 b** results in the derivation tree depicted in **Figure 4.2**. Initially, the rule [*Class*0] is applied to instance A0, which is a projection of instance A. For A0 to be translated roleA0, the instance representing A0's single role, must be translated. Thus, the premise of the application of [*Class*0] translates roleA0 by applying rule [*Role*0]. For roleA0 to be translated, A1 must be translated. Thus, the premise of the application of [*Role*0] translates A1 by applying rule [*Class*1]. The resulting C++ AST generated by this derivation tree is depicted in **Figure 4.3**[1].

[1]The authors are aware that the code generated by this AST is nonsensical. This is because the NDS example, which was selected to illustrate using NDS, is also nonsensical.

$$\dfrac{\dfrac{}{\mathsf{Class1}(\text{“}A''\text{)} \mapsto \mathsf{CppClassForwardDec}(\text{“}A''\text{)}}\,[Class1]}{\mathsf{Role0}(\text{“}roleA''\text{, }\mathsf{Class1}(\text{“}A''\text{)) } \mapsto \mathsf{FunctionMember}(\text{“}roleA''\text{, }\mathsf{CppClassForwardDec}(\text{“}A''\text{))}\,[Role0]}$$

$$\dfrac{\mathsf{Role0}(\text{“}roleA''\text{, }\mathsf{Class1}(\text{“}A''\text{))} \mapsto \mathsf{FunctionMember}(\text{“}roleA''\text{, }\mathsf{CppClassForwardDec}(\text{“}A''\text{))}}{\mathsf{Class0}(\text{“}A''\text{, }\mathsf{Role0}(\text{“}roleA''\text{, }\mathsf{Class1}(\text{“}A''\text{))}) \mapsto \mathsf{CppClass}(\text{“}A''\text{, }\mathsf{FunctionMember}(\text{“}roleA''\text{, }\mathsf{CppClassForwardDec}(\text{“}A''\text{))})}\,[Class0]$$

Figure 4.2: Example Derivation applying NDS

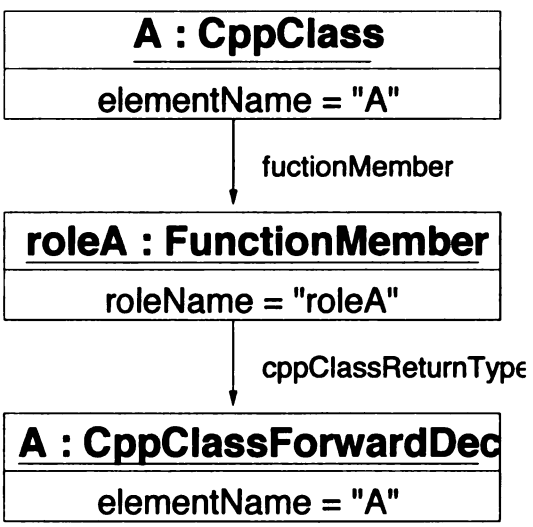


Figure 4.3: C++ AST

Notice, the derivation is finite as are all derivations that use this set of rules. Hence, this small example achieves our goal of specifying a compositional semantic function that maps ASG in the syntactic domain of a non-hierarchical language (UML) to elements of the semantic domain of another language (C++).

4.2 Optimization

We optimize the process of translating an ASG to a target-language AST by eliminating the explicit construction of a projection AST, which currently is constructed and then discarded. Recall, projection types are organized into layers, where different projection types in the same layer do not have the same source expression type. This has allowed us to informally define *proj*, an indexed family of functions (one function per layer). Each function translates ASGs to projection ASTs (**Section 3**). More formally, we define $proj(i)$ (where $i \in \mathcal{I}$) as a partial function that translates ASGs to projection ASTs, where each projection AST is an instance of a class in $Layer_{\mathcal{P}}(i)$. Next, we used NDS to specify *trans*, a semantic function that translates projection ASTs to target ASTs (**Section 4.1**). For each i, the composition of *trans* and $proj(i)$ is a partial function that translates ASGs to target language ASTs. Our key insight is that it is possible to define a family of functions, *transproj*, where each function translates ASGs to target language ASTs. Specifically, there exists an indexed family of functions *transproj*, such that for all $i \in dom(proj)$, $transproj(i) = trans \circ proj(i)$. In this section we construct *transproj* and in doing so optimize the translation process by eliminating the explicit construction of projection ASTs.

Unfortunately, NDS is designed to specify only one function and hence cannot specify a family of functions. Thus, to specify *transproj*, we extended NDS into a new representation called, *layered natural deduction systems* (LNDS). For clarity, we refer to the rules in an LNDS as *expression inference rules.*

$$\frac{roleClass \quad \mapsto_0 \quad functionMember}{\begin{array}{l}\mathsf{Class}(elementName, roleClass) \\ \quad \mapsto_0 \\ \mathsf{CppClass}(elementName, functionMember)\end{array}} \; [ClassAtLayer0]$$

$$\frac{}{\begin{array}{l}\mathsf{Class}(elementName) \\ \quad \mapsto_1 \\ \mathsf{CppClassForwardDec}(elementName)\end{array}} \; [ClassAtLayer1]$$

$$\frac{roleClass \quad \mapsto_1 \quad cppClassForwardDec}{\begin{array}{l}\mathsf{Role}(roleName, roleClass) \\ \quad \mapsto_0 \\ \mathsf{FunctionMember}(roleName, cppClassForwardDec)\end{array}} \; [RoleAtLayer0]$$

Figure 4.4: Example LNDS

An LNDS comprises one or more *rule layers*, where a rule layer is a set of expression inference rules that collectively specify a function in *transproj*. Syntactically, we indicate the rule layer to which a rule belongs by appending the name of the function to the maplet symbol in the conclusion. The symbol takes the form $\mapsto_k$, where k is the function name. The premises of an expression inference rule can refer to a different function than that specified by the conclusion. To clarify, we append the name of the function used to apply the premise to the premise's maplet symbol.

Figure 4.4 depicts an LNDS specification of a semantic function that assigns meaning to instances of Class and Role. The maplet symbol in each conclusion is appended with the name of the function specified by the rule. Specifically, expression rules [*ClassAtLayer*0] and [*RoleAtLayer*0] specify function 0; whereas expression rule [*ClassAtLayer*1] specifies function 1. The maplet symbols in the premises have been labeled to indicate which function to apply to translate the premise's subject. The maplet symbol in the premise of [*ClassAtLayer*0] is appended with 0. Similarly, the premise of [*RoleAtLayer*0] is appended with 1.

4.3 Procedure for generating an LNDS

We define the procedure for constructing an LNDS from an NDS that operates over projection types and the code for these projection types. To generate an expression rule that operates over labeled graphs from a projection rule that operates over terms, we let the subject operator of the generated expression rule be the source expression type of the projection rule's subject operator. The names of the parameters of the subject operator of an expression rule are the names of the parameters of the subject operator in the projection rule.

Our NDS specifies only one function; whereas our LNDS specifies a family of functions. We infer which function a generated expression rule specifies using the suffix of the subject operator of the projection rule. Recall, this suffix is the name of a layer. We append this to the maplet symbol in the conclusion of the expression rule. We infer which function to apply to translate the premise subject of an expression rule using the signature of the operation in the projection type that bears its name. Specifically, the signature of the operation will include a return type, which will be the projection type of which the operation will return an instance. The name of this projection type will contain a layer name that we append the maplet symbol the corresponding expression rule premise.

Figure 4.4 depicts the LNDS equivalent to the NDS shown in **Figure 4.1**. Specifically, [*ClassAtLayer*0], [*ClassAtLayer*1], and [*RoleAtLayer*0], (**Figure 4.4**) correspond to [*Class*0], [*Class*1] and [*Role*0] (**Figure 4.1**) respectively. The subject operator of each expression rule's conclusion is the source expression type of the subject operator of the corresponding projection rule. Specifically, the subject operator of expression rules [*ClassAtLayer*0] and [*ClassAtLayer*1] is `Class`, which is the source expression type of `Class0` and `Class1`, the subject operators of their respective projection rules. The subject operator of expression rule [*RoleAtLayer*0] is `Role`, which is the source expression type of `Role0`, the subject operator of projection rule

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Class}(\text{“}A''\text{)} \mapsto_1 \mathsf{CppClassForwardDec}(\text{“}A''\text{)}}\,[ClassAtLayer1]}{\mathsf{Role}(\text{“}roleA'', \mathsf{Class}(\text{“}A''\text{)}) \mapsto_0 \mathsf{FunctionMember}(\text{“}roleA'', \mathsf{CppClassForwardDec}(\text{“}A''\text{)})}\,[RoleAtLayer0]}{\mathsf{Class}(\text{“}A'', \mathsf{Role}(\text{“}roleA'', \mathsf{Class}(\text{“}A''\text{)})) \mapsto_0 \mathsf{CppClass}(\text{“}A'', \mathsf{FunctionMember}(\text{“}roleA'', \mathsf{CppClassForwardDec}(\text{“}A''\text{)}))}\,[ClassAtLayer0]$$

Figure 4.5: Example Derivation applying LNDS

[*Role*0]. The names of the parameters of the subject operator of each expression rule are those of the corresponding projection rule's subject operator. Observe, we are able to infer the layer name to append to each maplet symbol in the premises and conclusions of the expression rules using the process described above.

The expression rules in **Figure 4.4** are applied to ASGs, rather than ASTs. **Figure 4.5** depicts the derivation tree that results from applying the LNDS shown in **Figure 4.4** to the ASG depicted in **Figure 3.4 a**. First, the rule [*ClassAtLayer*0] is applied to instance A. For A to be translated, roleA (the instance representing A's single role) must be translated under function 0. Thus, the premise of the application of [*ClassAtLayer*0] is the result of applying rule [*RoleAtLayer*0] to roleA. For roleA to be translated under function 0, A must be translated under function 1. This is accomplished in the premise of the application of rule [*RoleAtLayer*0]. This derivation produces the same C++ AST (depicted in **Figure 4.3**) as the derivation shown in **Figure 4.2**.

4.4 Benefit: Symmetric Premise

A benefit of using LNDS is that we are able to apply different functions to the same ASG. An example where this ability is useful is translating a role in UML,

$$\frac{this \quad \mapsto_{type} \quad returnType}{\mathsf{Role}(roleName, roleClass) \mapsto_{impl} \mathsf{AssocRole}(roleName, returnType)} \; [\mathsf{Role}_{impl}]$$

Figure 4.6: Example of Symmetric Premises

which represents the part an instance of a class plays in an association or aggregation. Because we use roles as a way to reference a specific instance of a class that participates in the relationship, we represent roles in C++ as a private data member and a public function member. The data member stores a pointer to an instance of a C++ class and the function member provides a method for retrieving the pointer to the instance of a C++ class. An AssocRole is a function member and a data member that represent a UML role. The name of the data member is that of the role and is transmitted in string *roleName*. Similarly, the name of the function member is that of the role prepended with the keyword "get". The data type of the data member (which is the same as the return type of the function member) is constructed by translating the role using a different function.

The syntactic structure that facilitates this ability is a *symmetric premise*, i.e., an expression rule premise whose subject is the subject of the rule's conclusion. Continuing with our example, the instance of role is the subject of the conclusion and the subject of the premise, which constructs the return type. We indicate a premise is a symmetric premise by using the keyword *this* as the subject of the premise.

Figure 4.6 depicts an inference rule that specifies the translation of role described earlier. Specifically, it translates an instance of Role to an instance of AssocRole. The return type of the data member and function member, which constitute AssocRole, is constructed by the symmetric premise.

4.5 Formally Representing LNDS

In the sequel, our algorithms determine if an LNDS is well-formed by appealing to the metamodel of the source language. Hence we need to represent an LNDS as a datatype. We have chosen to formally specify the LNDS data type in Z to correspond with the rest of our work.

Fundamentally, an LNDS is a set of inference rules and axioms. To model an LNDS, we introduce the following given sets.

$$[FV\_ID, L\_ID]$$

These sets represent the identifiers for free variables and layers respectively. As with the other identifiers we introduced, elements of these sets can only be compared for equality. We declare the special free variable *this* to be an element of FV_ID.

$$| \; this : FV\_ID$$

A free variable in any LNDS (for example **Figure 4.6**) corresponds an element of FV_ID. To refer to a specific FV_ID, we use render the name of the corresponding free variable in italics (as this is how the free variable names are represented in the rules). For example, *roleName* refers to the free variable identifier that corresponds to the free variable of the same name in **Figure 4.6**. We adopt a similar convention for elements of the set L_ID.

A free variable is associated with a string representing its name.

$$FREEVARIABLE == STRING$$

We also define accessor functions *getFV* and *getFVName*, where *getFV* returns the free variable that corresponds to a given FV_ID and *getFVName* returns the string representing the free variables name when given a free variable.

$$getFV : FV\_ID \rightarrow FREEVARIABLE$$
$$getFVName : FREEVARIABLE \rightarrow STRING$$

We model a layer as a tuple consisting of a string, which is the name of a function, and a natural number, which is the index of the layer.

$$LAYER == STRING \times \mathbb{N}$$

For example, ("impl", 0) and ("type", 1) represent the layers used by the LNDS depicted in **Figure 4.6**. The accessor function *getLayer* returns the layer tuple that corresponds to a given *L_ID*. The accessor functions *getLayerName* and *getLayerIndex* return a string that is the layer's name and a natural number, which is the layer's index, respectively. These functions are formalized as follows.

$$getLayer : L\_ID \rightarrow LAYER$$
$$getLayerName : LAYER \rightarrow STRING$$
$$getLayerIndex : LAYER \rightarrow \mathbb{N}$$
$$\hline$$
$$getLayerName = first[STRING, \mathbb{N}]$$
$$getLayerIndex = second[STRING, \mathbb{N}]$$

For example, applying the function *getLayer* to *impl* yields the tuple ("impl", 0). Applying the function *getLayerName* to this tuple yields "impl" and applying *getLayerIndex* to the same tuple yields 0.

We model a premise as a tuple consisting of two free variables (the source and the target) and a layer identifier.

$$PREMISE == FV\_ID \times L\_ID \times FV\_ID$$

For example, the premise of rule [Role_{impl}] is formalized as the tuple (*this*, *type*, *returnType*). We define three accessor functions to return the first, second and third elements of a tuple representing a premise.

$$
\begin{array}{|l}
getPL : PREMISE \rightarrow L\_ID \\
getPS : PREMISE \rightarrow FV\_ID \\
getPT : PREMISE \rightarrow FV\_ID \\
\hline
\forall fv1, fv2 : FV\_ID;\ layer : L\_ID \bullet \\
\quad getPS(fv1, layer, fv2) = fv1 \\
\quad \land \\
\quad getPL(fv1, layer, fv2) = layer \\
\quad \land \\
\quad getPT(fv1, layer, fv2) = fv2
\end{array}
$$

Applying function *getPL* to the tuple (*this*, *type*, *returnType*) yields *type*. Applying functions *getPS* and *getPT* to tuple yields *this* and *returnType* respectively.

We model an AST schema as a class identifier and a sequence of free variable identifiers. Because we require all operators to name a class in the metamodel, each operator is represented as a *CLASS_ID*[2].

$$ASTSCHEMA == CLASS\_ID \times \text{seq}\, FV\_ID$$

For example, the target of rule $[\mathsf{Role}_{impl}]$ is represented by the tuple (**AssocRole**, $\langle roleName,\ returnType \rangle$). The accessor functions *getOperator* and *getFreeVariables* returns the class identifier and free variable identifiers of an AST schema tuple respectively.

$$
\begin{array}{|l}
getOp : ASTSCHEMA \rightarrow CLASS\_ID \\
getFVIDs : ASTSCHEMA \rightarrow \text{seq}\, FV\_ID \\
\hline
getOp = first[CLASS\_ID, \text{seq}\, FV\_ID] \\
getFVIDs = second[CLASS\_ID, \text{seq}\, FV\_ID]
\end{array}
$$

An inference rule's conclusion is a tuple consisting of two AST schemas (the subject and the target) and a layer identifier.

$$CONCLUSION == ASTSCHEMA \times L\_ID \times ASTSCHEMA$$

[2]The target language operators will be depicted as classes on the target language metamodel. Owing to space constraints, we do not depict the C++ metamodel.

For example, we represent the conclusion of rule [Role_{impl}] with the tuple ((Role, $\langle roleName, roleClass \rangle$), *impl*, (AssocRole, $\langle roleName, returnType \rangle$)). We define three accessor functions to access the first, second and third elements of a tuple representing a conclusion.

$$
\begin{array}{|l}
getCS : CONCLUSION \rightarrow ASTSCHEMA \\
getCT : CONCLUSION \rightarrow ASTSCHEMA \\
getCL : CONCLUSION \rightarrow L\_ID \\
\hline
\forall\, a1, a2 : ASTSCHEMA;\ layer : L\_ID \bullet \\
\quad getCS(a1, layer, a2) = a1 \\
\quad \wedge \\
\quad getCL(a1, layer, a2) = layer \\
\quad \wedge \\
\quad getCT(a1, layer, a2) = a2
\end{array}
$$

Applying the *getCS* function to the tuple ((Role, $\langle roleName, roleClass \rangle$), *impl*, (AssocRole, $\langle roleName, returnType \rangle$)) yields (Role, $\langle roleName, roleClass \rangle$). Applying *getCT* to the same tuple yields (AssocRole, $\langle roleName, returnType \rangle$). Lastly, applying *getCL* yields *impl*.

We represent an inference rule as a schema consisting of a set of premises and a conclusion. The four invariants constrain the legal schema bindings as follows. Each parameter of the conclusion's target is either a parameter of the conclusion's subject or is a premise's target. Each premise's target is a parameter of the target of the conclusion. Each premise's subject is a free variable that parameterizes the subject of the conclusion or is a free variable whose name is the keyword *this*. The layers used in all of the premises are constrained in that their index must be greater than or equal to the index of the layer in the conclusion.

InferenceRule

$$
\begin{array}{l}
prems : \mathbb{P}\, PREMISE \\
conc : CONCLUSION \\
\hline
\mathrm{ran}(getFVIDs(getCT(conc))) \\
\subseteq \\
\mathrm{ran}(getFVIDs(getCS(conc))) \cup getPT(\!|\ prems\ |\!) \\
getPT(\!|\ prems\ |\!) \subseteq \mathrm{ran}(getFVIDs(getCT(conc))) \\
getPS(\!|\ prems\ |\!) \subseteq \mathrm{ran}(getFVIDs(getCS(conc))) \cup \{this\} \\
\forall\, p : prems \bullet \\
\quad (getPL \fatsemi getLayer \fatsemi getLayerIndex)(p) \leq \\
\quad (getCL \fatsemi getLayer \fatsemi getLayerIndex)(conc)
\end{array}
$$

Chapter 5: LNDS Example

We present an LNDS and briefly illustrate the process of generating C++ ASTs from UML ASGs. Because generating code is not the thrust of this thesis, our example is small and aims only to give a flavor of how code generation is performed.

Figures 5.1 - 5.3 depict an LNDS specification of a semantic function that translates UML ASGs to C++ ASTs. This example has two rule layers named *type* (**Figure 5.1**) and *impl* (**Figure 5.2** and **Figure 5.3**), where *impl* < *type*. This semantic function translates UML classes to C++ classes, UML attributes and associations to C++ function members and data members, and UML generalizations to C++ class inheritance. Briefly, we explain each rule.

5.1 Type layer

First, we describe the rules comprising the *type* rule layer, which is depicted in **Figure 5.1**. Axiom [Class_{type}] is used to construct an instance of CppClass, which represents a C++ class declaration. When CppClass is instantiated with only a name, the resulting instance of CppClass is a forward declaration. All applications of [Class_{type}] generate forward declarations. Axiom [Domain_{type}] constructs a C++ domain from a UML domain. We use domains to represent primitive types (i.e., non-class types such as boolean and integer).

Rule [Role_{type}] is used to construct an AST representation of a pointer to an instance of a C++ class, from the information contained within a UML Role. Its premise computes the C++ class declaration (i.e., the instance of CppClass) associated with the UML class referenced in the role.

Rule [$\mathsf{Parameter}_{type}$] is used to construct an AST representation of a parameter of a function member. The UML parameter consists of a name and a type. The C++

$$\frac{}{\mathsf{Class}(elementName)} \; [\mathsf{Class}_{type}]$$
$$\mapsto_{type}$$
$$\mathsf{CppClass}(elementName)$$

$$\frac{}{\mathsf{Domain}(domainName)} \; [\mathsf{Domain}_{type}]$$
$$\mapsto_{type}$$
$$\mathsf{CppDomain}(domainName)$$

$$\frac{roleClass \mapsto_{type} roleCppClass}{\mathsf{Role}(roleName, roleClass)} \; [\mathsf{Role}_{type}]$$
$$\mapsto_{type}$$
$$\mathsf{PointerType}(roleName, roleCppClass)$$

$$\frac{paramType \mapsto_{type} cppType}{\mathsf{Parameter}(parameterName, paramType)} \; [\mathsf{Parameter}_{type}]$$
$$\mapsto_{type}$$
$$\mathsf{CppParameter}(parameterName, cppType)$$

$$\frac{\begin{array}{l} returnType \mapsto_{type} dataType \\ opParams \mapsto_{type} paramTypes \end{array}}{\begin{array}{l} \mathsf{Operation}(operationName, returnType, \\ \qquad opParams) \end{array}} \; [\mathsf{Operation}_{type}]$$
$$\mapsto_{type}$$
$$\mathsf{OperationType}(operationName, dataType, paramTypes)$$

Figure 5.1: LNDS Example (1)

$$\frac{attributeDomain \quad \mapsto_{type} \quad type}{\begin{array}{l}\textsf{Attribute}(attributeName, attributeDomain) \\ \quad \mapsto_{impl} \\ \textsf{DataMember}(attributeName, type)\end{array}} \; [\textsf{Attribute}_{impl}]$$

$$\frac{this \quad \mapsto_{type} \quad returnType}{\begin{array}{l}\textsf{Role}(roleName, roleClass) \\ \quad \mapsto_{impl} \\ \textsf{AssocRole}(roleName, returnType)\end{array}} \; [\textsf{Role}_{impl}]$$

Figure 5.2: LNDS Example (2)

parameter's name will be that of the UML parameter (*parameterName*). The C++ parameter's type will be constructed in the premise.

Rule [$\textsf{Operation}_{type}$] is used to construct an AST representation of the declaration of a function member. The function member's name will be that of the UML operation (*operationName*). The function member's return type will be constructed by the first premise. The function member's parameter list will be constructed by the second premise.

5.2 Impl layer

Next, we describe the rules comprising the *impl* rule layer, which is depicted in **Figure 5.2** and **Figure 5.3**. Rule [$\textsf{Attribute}_{impl}$] constructs an AST representation of a C++ class data member. The data member's name is that of the UML attribute (*attributeName*) and its type is constructed in the premise.

Rule [$\textsf{Role}_{impl}$] is used to translate a UML role. Rule [$\textsf{Aggregation}_{impl}$] constructs an AST representation of a C++ **Aspect**, which is a construct that we invented to represent a set of sets of function members and data members that, when combined

$$\frac{part \quad \mapsto_{impl} \quad assocRole}{\begin{array}{l}\mathsf{Aggregation}(elementName, part) \\ \quad \mapsto_{impl} \\ \mathsf{Aspect}(elementName, assocRole)\end{array}} \; [\mathsf{Aggregation}_{impl}]$$

$$\frac{associateRoles \quad \mapsto_{impl} \quad assocRoleSet}{\begin{array}{l}\mathsf{OrdinaryAssociation}(elementName, \\ \qquad\qquad associateRoles) \\ \quad \mapsto_{impl} \\ \mathsf{Aspect}(elementName, assocRoleSet)\end{array}} \; [\mathsf{OrdinaryAssociation}_{impl}]$$

$$\frac{superClass \quad \mapsto_{type} \quad cppSuper}{\begin{array}{l}\mathsf{Generalization}(superClass) \\ \quad \mapsto_{impl} \\ \mathsf{CppGeneralization}(cppSuper)\end{array}} \; [\mathsf{Generalization}_{impl}]$$

$$\frac{\begin{array}{rcl} classAtts & \mapsto_{impl} & dataMems \\ opClass & \mapsto_{type} & cppOps \\ /assocs & \mapsto_{impl} & assocAspect \\ /aggs & \mapsto_{impl} & aggAspect \\ supClass & \mapsto_{impl} & cppSupers \end{array}}{\begin{array}{l}\mathsf{Class}(elementName, \\ \qquad supClass, classAtts, \\ \qquad opClass, /assocs, /aggs) \\ \quad \mapsto_{impl} \\ \mathsf{CppClass}(elementName, \\ \qquad cppSupers, dataMems, \\ \qquad cppOps, flattten(assocAspect), \\ \qquad flatten(aggAspect)))\end{array}} \; [\mathsf{Class}_{impl}]$$

Figure 5.3: LNDS Example (3)

and pretty printed, represent all of the code required to implement a UML ordinary association or aggregation. We chose to represent a UML aggregation in C++ as a one way relationship in which the instance of the class playing the role of the whole maintains information about the instance of the class playing the role of the part[1]. The *part* role is translated by the premise.

Rule [OrdinaryAssociation$_{impl}$] constructs an AST representation of an Aspect for each UML ordinary association. The single premise, which translates a Role to a set of function and data members, is applied to each Role in an association. Thus, it translates a set of roles to a set of sets of function and data members.

Rule [Generalization$_{impl}$] is used to translate a UML generalization to an AST representation of a C++ inheritance relation. The single premise translates the UML super class in the generalization to a C++ class declaration.

Lastly, rule [Class$_{impl}$] translates a UML class to a C++ class declaration. Notice, rule [Class$_{type}$] also did this, but without any information about data members, function members, or generalizations. If a C++ class declaration has already been constructed with a given name, this rule will elaborate that declaration with new information, rather than construct a new class. A C++ class consists of a sequence of members (both function and data) and generalizations. The premises of this rule translate a UML class's attributes, operations, associations, aggregations and generalizations to C++ function member declarations, data members, and inheritance relationships between C++ classes. The *flatten* function is applied to *assocAspect* (and *aggAspect*) to change it from a set of sets of function and data members to a set of function and data members.

5.3 Example derivation

Figure 5.4 depicts the derivation tree that translates class A of the UML class

[1]Because the whole role is not used by this translation it is not represented in the rule.

$$\frac{}{\mathsf{Class}(\text{“}B''\text{)}} \; [\mathsf{Class}_{type}]$$

$$\mapsto_{type}$$

$$\mathsf{CppClass}(\text{“}B''\text{)}$$

$$\frac{}{\mathsf{Role}(\text{“}roleB''\text{, Class}(\text{“}B''\text{))}} \; [\mathsf{Role}_{type}]$$

$$\mapsto_{type}$$

$$\mathsf{PointerType}(\text{“}roleB''\text{, CppClass}(\text{“}B''\text{))}$$

$$\frac{}{\mathsf{Role}(\text{“}roleB''\text{, Class}(\text{“}B''\text{))}} \; [\mathsf{Role}_{impl}]$$

$$\mapsto_{impl}$$

$$\mathsf{AssocRole}(\text{“}roleB''\text{, PointerType}(\text{“}roleB''\text{, CppClass}(\text{“}B''\text{)))}$$

$$\frac{}{\mathsf{Aggregation}(\text{“}AtoBAgg''\text{, Role}(\text{“}roleB''\text{, Class}(\text{“}B''\text{))))}} \; [\mathsf{Aggregation}_{impl}]$$

$$\mapsto_{impl}$$

$$\mathsf{Aspect}(\text{“}AtoBAgg''\text{, AssocRole}(\text{“}roleB''\text{, PointerType}(\text{“}roleB''\text{, CppClass}(\text{“}B''\text{))))}$$

$$\frac{}{\mathsf{Class}(\text{“}A''\text{, }\varnothing\text{, }\varnothing\text{, }\varnothing\text{, }\varnothing\text{, Aggregation}(\text{“}AtoBAgg''\text{, Role}(\text{“}roleB''\text{, Class}(\text{“}B''\text{))))}} \; [\mathsf{Class}_{impl}]$$

$$\mapsto_{impl}$$

$$\mathsf{CppClass}(\text{“}A''\text{, }\varnothing\text{, }\varnothing\text{, }\varnothing\text{, }\varnothing\text{, }\mathit{flatten}(\mathsf{Aspect}(\text{“}AtoBAgg''\text{, AssocRole}(\text{“}roleB''\text{, PointerType}(\text{“}roleB''\text{, CppClass}(\text{“}B''\text{))))))$$

Figure 5.4: LNDS Derivation Tree Example

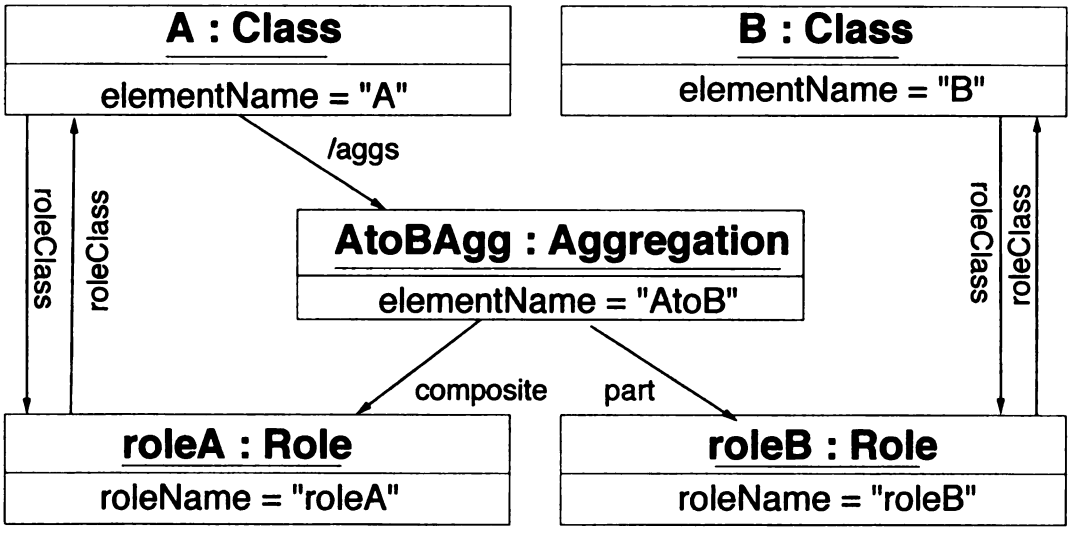


Figure 5.5: UML ASG

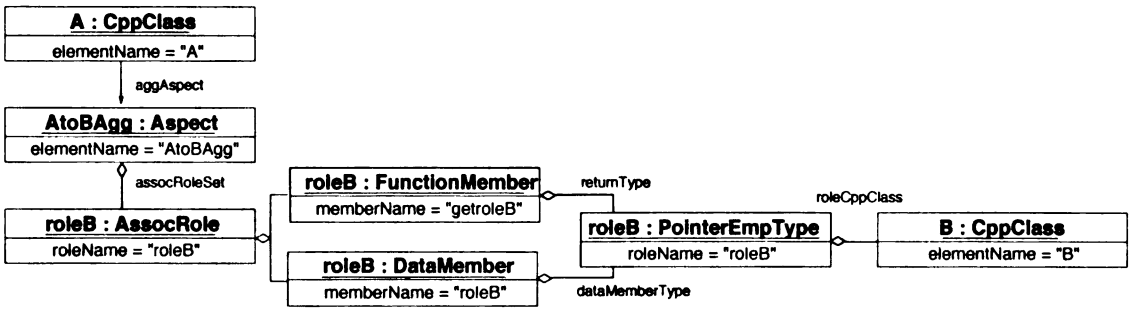


Figure 5.6: Constructed C++ AST

diagram depicted in **Figure 3.3**. The ASG for this class diagram is depicted in **Figure 5.5**. First, the rule [Class_{impl}] is applied to instance A. For A to be translated, aggregation AtoBAgg must be translated. Thus, the single premise shown of the application of [Class_{impl}] is the result of applying rule [$\mathsf{Aggregation}_{impl}$] to AtoBAgg[2]. For AtoBAgg to be translated, the part role, roleB must be translated. Thus, the single premise of the application of [$\mathsf{Aggregation}_{impl}$] is the result of applying rule [Role_{impl}] to roleB. For roleB to be translated, a type for it must be constructed. Hence, the single premise of the application of [Role_{impl}] is the result of applying rule [Role_{type}] to roleB. For the type of roleB to be constructed the instance of its class, B, must be translated. Thus, the single premise of the application of [Role_{type}] translates B into a forward declaration of the C++ class B.

The resulting C++ AST is depicted in **Figure 5.6**. **Figure 5.7** depicts the C++ code that results from pretty printing this AST . Specifically, both instance A and instance B are represented as C++ classes. Aggregation AtoBAgg is represented as a function member and data member in class A, where the data member stores a pointer to an instance of class B and the function member accesses this pointer.

[2]The other premises of the [Class_{impl}] rule are unnecessary for this translation and hence have been elided.

```
class A {
   public:
      const B * getAtoBAgg();

   protected:
      const B * AtoBAgg;
   }:


class B {
   };
```

Figure 5.7: Code resulting from application of rule Class_{impl}

Chapter 6: Validation

An LNDS is well-formed if it is impossible to construct an infinite derivation by applying its rules. To ensure an LNDS is well-formed, we developed a tool that determines if an LNDS is well-formed when given an LNDS and a metamodel depicting the (original) abstract syntax of the (non-hierarchical) language.

6.1 Automatically checking if an LNDS is well-formed

Each derivation traverses an ASG and applies functions to translate its nodes and produce target language ASTs. A derivation is guaranteed to be finite if the traversal never applies the same function to an ASG node more than once. We verify this property by marking the traversal path that corresponds to each function in the LNDS (as specified by one layer) on the metamodel and checking that the path is acyclic. This is possible because the links, which are taken to traverse an ASG, are depicted as associations in the metamodel and the ASG node types are depicted as classes in the metamodel. Although the algorithms are specified here in Z, we have implemented them in Haskell [23, 4].

To illustrate our algorithms we introduce a trivial non-hierarchical language HIJ whose abstract syntax is depicted in **Figure 6.1**. An LNDS specified semantic function, which assigns meaning to syntactically valid programs in the HIJ language, is depicted in **Figure 6.2**. Observe, there are two functions, *Zero* and *One* each of which is specified by a rule layer and where the index of rule layer *Zero* is less than the index of rule layer *One*. Although we demonstrate our algorithms by applying them to layer *Zero* of the HIJ language, we have used these algorithms to ensure the rules presented in **Chapter 5** are well-formed.

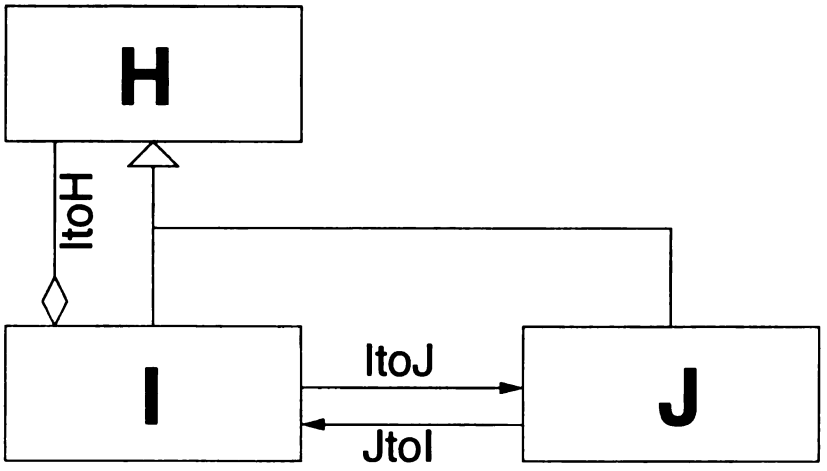


Figure 6.1: HIJ language

$$\frac{\begin{array}{ll} ItoH & \mapsto_{Zero} \; TransOfIH \\ ItoJ & \mapsto_{One} \; TransOfIJ \end{array}}{\begin{array}{l} I(ItoH, ItoJ) \\ \quad \mapsto_{Zero} \\ TranslationOfIZero(TransOfIH, TransOfIJ) \end{array}} [IZero]$$

$$\frac{JtoI \; \mapsto_{Zero} \; TransOfJI}{\begin{array}{l} J(JtoI) \\ \quad \mapsto_{Zero} \\ TranslationOfJZero(TransOfJI) \end{array}} [JZero]$$

$$\frac{}{\begin{array}{l} I() \\ \quad \mapsto_{One} \\ TranslationOfIOne() \end{array}} [IOne]$$

Figure 6.2: HIJ LNDS

6.1.1 Adding Marked Associations

The first step in checking if an LNDS rule layer is well-formed is adding marked associations to the metamodel to depict the traversal path. Specifically, we add a marked association connecting the class depicting the subject operator with each class depicting the type of a premise subject. We have designed our LNDS so that the name of the free variable in the premise's subject corresponds to the name of an association connecting the class depicting the subject operator to the class depicting the type of the premise's subject in the metamodel[1]. If the unmarked association is an aggregation, we add a marked aggregation; otherwise we add a marked ordinary association.

Below we depict the Z schema for representing a marked metamodel, *MarkedMetamodel*, which we then initialize by adding marked associations using operation *InitializeMarkedMetamodel*. A *MarkedMetamodel* extends a *Metamodel* with a relation named *markedAggs*, which relates two class identifiers, and a relation named *markedAssocs*, which also relates two class identifiers. We restrict the domain and range of these relations to be subsets of the class identifiers that comprise the *Metamodel*.

MarkedMetamodel

$$
\begin{array}{l}
Metamodel \\
markedAggs : CLASS\_ID \rightarrow CLASS\_ID \\
markedAssocs : CLASS\_ID \rightarrow CLASS\_ID \\
\hline
\mathrm{dom}\ markedAggs \subseteq classes \\
\mathrm{ran}\ markedAggs \subseteq classes \\
\mathrm{dom}\ markedAssocs \subseteq classes \\
\mathrm{ran}\ markedAssocs \subseteq classes
\end{array}
$$

We initialize a *MarkedMetamodel* with operation *InitializeMarkedMetamodel* using information from the LNDS. The inputs are a set of inference rules and the

[1]We adopted this convention to simplify the implementation. It is not necessary in general.

identifier of the layer we are interested in marking.

$$
\begin{array}{|l}
\textit{InitializeMarkedMetamodel} \\
\hline
\Delta \textit{MarkedMetamodel} \\
\textit{rules?} : \mathbb{P}\, \textit{InferenceRule} \\
\textit{layer?} : \textit{L\_ID} \\
\hline
\textit{markedAggs}' = \{c1, c2 : \textit{CLASS\_ID} \mid \\
\quad \exists\, a : \textit{assocs};\ r : \textit{rules?};\ p : \textit{PREMISE} \mid p \in r.\textit{prems} \bullet \\
\qquad \textit{getFV}(\textit{getPS}(p)) = \textit{getAssocName}(\textit{getAssoc}(a)) \\
\qquad \land \\
\qquad \textit{getAssocType}(\textit{getAssoc}(a)) = \textit{aggregation}(c1, c2) \\
\qquad \land \\
\qquad \textit{getCL}(r.\textit{conc}) = \textit{getPL}(p) = \textit{layer?}\} \\
\textit{markedAssocs}' = \{c1, c2 : \textit{CLASS\_ID} \mid \\
\quad \exists\, a : \textit{assocs};\ r : \textit{rules?};\ p : \textit{PREMISE} \mid p \in r.\textit{prems} \bullet \\
\qquad \textit{getFV}(\textit{getPS}(p)) = \textit{getAssocName}(\textit{getAssoc}(a)) \\
\qquad \land \\
\qquad \textit{getAssocType}(\textit{getAssoc}(a)) = \textit{directed}(c1, c2) \\
\qquad \land \\
\qquad \textit{getCL}(r.\textit{conc}) = \textit{getPL}(p) = \textit{layer?}\} \\
\textit{classes}' = \textit{classes} \\
\textit{assocs}' = \textit{assocs} \\
\hline
\end{array}
$$

An element is added to *markedAggs* for every association where the name of the free variable in the premise's subject is the same as the name of the association, the type of the association is aggregation, and the layer index of the conclusion is the same as both the layer index we are marking and the layer index of the premise. The element we add to *markedAggs* is a tuple consisting of the class identifiers of the whole and part class in the aggregation. Similarly, an element is added to *markedAssocs* for every association where the name of the free variable in the premise's subject is the same as the name of the association, the type of the association is directed, and the layer index of the conclusion is the same as both the layer index we are marking and the layer index of the premise. The element we add to *markedAssocs* is a tuple consisting of the class identifiers of the source and target class in the ordinary association.

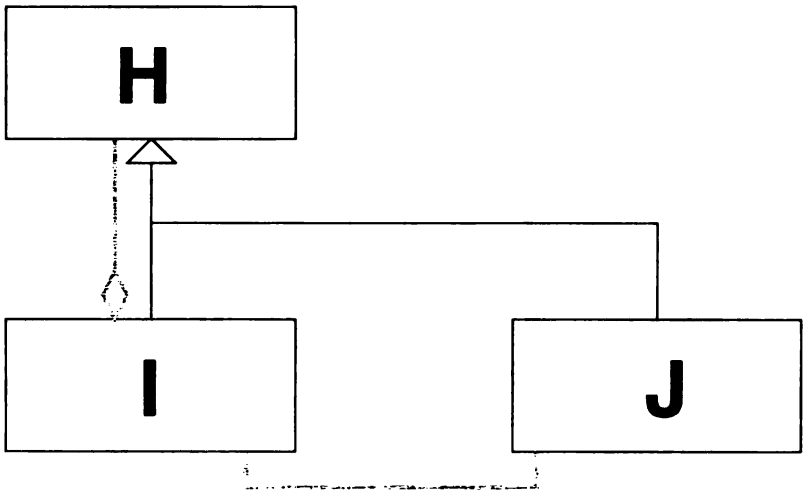


Figure 6.3: HIJ language after adding marked associations

For example, we initialize a *MarkedMetamodel* for our HIJ example. The initialized marked metamodel is depicted in **Figure 6.1** and inference rules [*IZero*] and [*JZero*] are depicted in **Figure 6.2**. Rule [*IZero*] has two premises. The first premise $ItoH \mapsto_{Zero} TransOfIH$ belongs to the same rule layer as the conclusion (0), hence a marked association depicting the relationship between the source operator, I, and the type of the premise, H, should be added to the metamodel. The association that depicts this relationship is an aggregation; thus, we add a marked aggregation that connects I to H. The next premise $ItoJ \mapsto_{One} TransOfJ$ is of a different rule layer, so we ignore it. This process is repeated for rule [*JZero*]. The marked metamodel consisting of classes, generalizations and marked associations (which are depicted in gray) resulting from the application of this algorithm is depicted in **Figure 6.3**[2].

6.1.2 Collapsing Generalizations

Generalizations obscure the cycles within the traversal path. Thus, we collapse all generalizations by replacing each association that involves the superclass of the generalization with associations involving each of the subclasses of the generalization.

[2]Because our algorithms concern themselves exclusively with marked aggregations and associations we have elided all unmarked associations and unmarked aggregations.

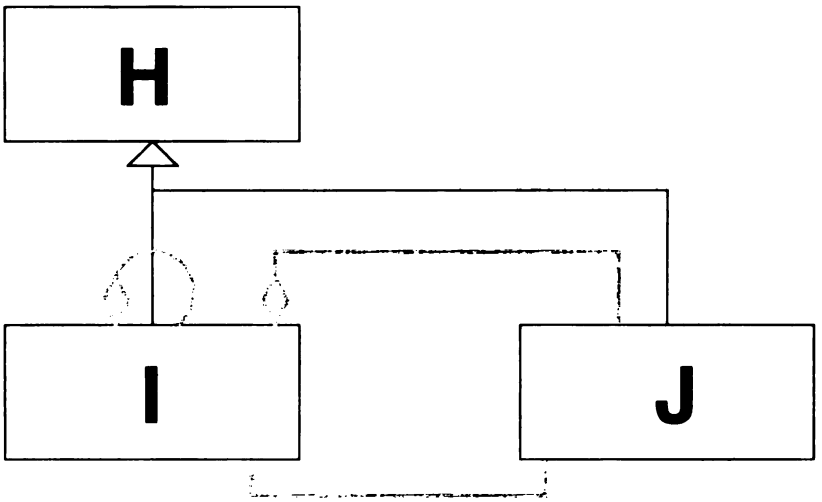


Figure 6.4: HIJ language after collapsing generalizations

The Z schema that collapses the generalizations of a *MarkedMetamodel* is formalized as follows.

CollapseGens

$\Delta MarkedMetamodel$

$classes' = classes$
$assocs' = assocs$
$markedAggs' = markedAggs \fatsemi (genRelation^+)$
$markedAssocs' = markedAssocs \fatsemi (genRelation^+)$

Figure 6.4 depicts the HIJ metamodel after the generalization relationship relating H (the superclass) to I and J (the subclasses) is collapsed. Originally, the marked aggregation ItoH related class I to class H. This marked aggregation has now been replaced with two marked aggregations that relate I to itself and I to J.

6.1.3 Eliminating Reflexive Aggregations

A *reflexive aggregation* is an association that aggregates a class to itself. Although these aggregations appear to be cycles within the traversal path, the definition of aggregation precludes an aggregation relating an instance to itself. Thus, they cannot

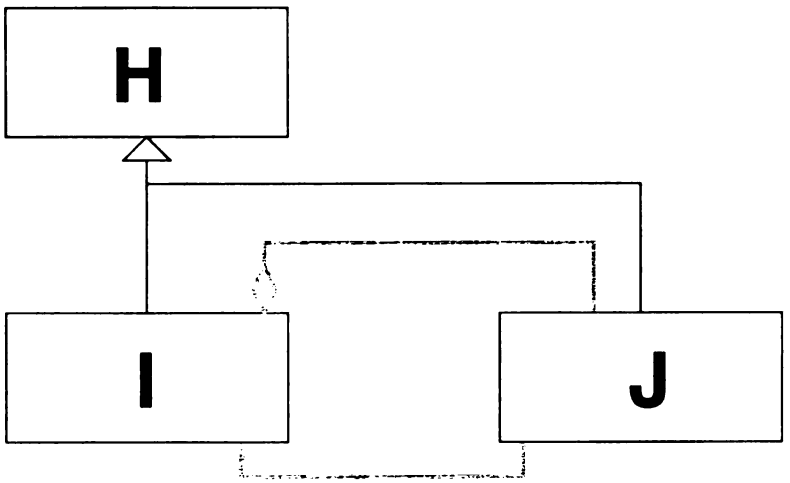


Figure 6.5: HIJ language after removing reflexive aggregations

specify cycles within a traversal and need to be eliminated prior to checking the traversal path (as depicted by the marked associations in the metamodel) for cycles. The Z schema that eliminates the reflexive aggregations in a marked metamodel is as follows.

EliminateReflexiveAggs

$\Delta MarkedMetamodel$

$classes' = classes$

$assocs' = assocs$

$markedAggs' = markedAggs \setminus \{c : CLASS\_ID \bullet (c, c)\}$

$markedAssocs' = markedAssocs$

Figure 6.4 depicts a reflexive aggregation relating class I to itself. Applying the *eliminateReflexiveAgg* procedure to this marked metamodel results in the marked metamodel depicted in **Figure 6.5** in which this aggregation has been removed.

6.1.4 Checking if a Graph is Cyclic

Our overall goal is to check if the traversal path specified by one rule layer is cyclic. To accomplish this we have added marked associations depicting the traversal

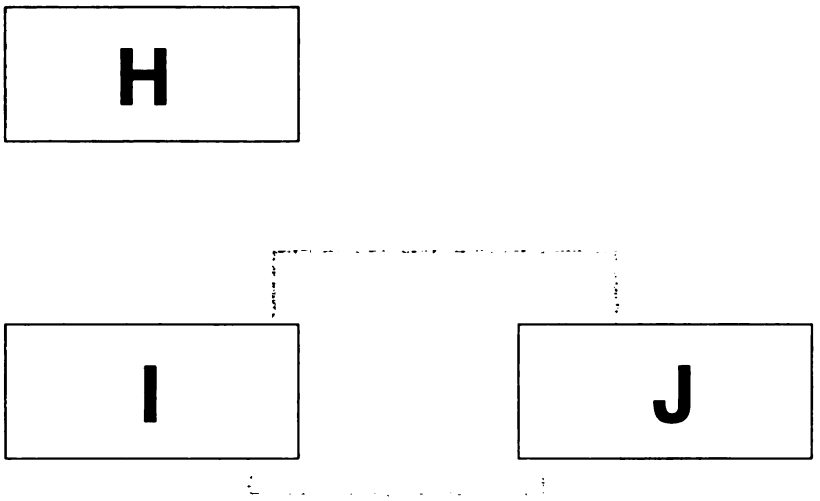


Figure 6.6: HIJ graph to be checked for cycles

path to the metamodel, collapsed generalizations and eliminated reflexive aggregations. To check the metamodel for cycles we must view it as a graph by viewing each class as a node and each marked association as a directed edge. Specifically, an aggregation is an edge that points from the composite class to the part class. Viewing a metamodel is this way reduces the problem to a cycle detection problem. If the marked edges, which depict the traversal path, form a cycle, the LNDS is ill-formed. This is formalized in Z as follows.

CyclicMarking

$$MarkedMetamodel$$

$$\exists c : CLASS\_ID \bullet (c, c) \in (markedAggs \cup markedAssocs)^+$$

Figure 6.6 depicts the graph representation of the HIJ metamodel. Observe that for clarity we have transformed aggregations to arrows that point from the container class to the part class[3]. It is now a trivial process to deduce that graph contains a cycle and thus the LNDS is not well-formed. If this graph did not contain a cycle, to prove that the LNDS was well-formed each of the above algorithms would have to be applied to the *One* rule layer.

[3]Ordinary associations have always been represented as arrows and hence are unmodified.

Chapter 7: Related Work

In this thesis, we have proposed a method for compositionally specifying UML's semantics using a syntactically extended version of NDS. By way of discussion, we discuss two other approaches to formally specifying UML's semantics.

First, we discuss *correspondence-style rules* (e.g., [33, 3, 8, 40, 41, 10, 39, 24, 26, 30, 9, 29]), which use natural language and code templates to specify the semantics for a portion of the UML notation. For example, [33], [17] and [18] translate UML diagrams to Z [35] and [3, 29] translate UML diagrams to PVS. [8, 40, 41] translate OMT [5], a precursor of UML, diagrams to LOTOS [6]. The primary shortcoming of correspondence-style rules is the imprecision of the prose specification, which hides ambiguities in the conditions for applying a rule. Thus, although this approach assigns semantic meaning to UML diagrams by translating the diagrams to programs, in a formal language, the ambiguity of the translation procedure makes it impossible to check if the the semantic specification is well-formed.

The second approach to assigning semantic meaning to UML diagrams is using *graph grammars*, which transform UML models to models in other domain specific languages, often times undergoing several intermediate transformations [28, 2, 1, 22]. The advantage to using intermediate transformations is that they are less complex than the direct transformation and hence are easier to specify and maintain. [28] specifies each transformation using a *domain mapping specification*, which is an extended version of a UML object diagram that depicts the relationships between objects in the source language and objects in the target language. [2, 1] specify each transformation with rules that depict mappings between metamodels and a sequencer, which tells the order in which the rule should be executed. The primary shortcoming of this approaches is that the transformations are specified with visual languages, whose semantics are informal. Thus, there is ambiguity in the transformation specification.

Again, it is impossible to check if each transformation is well-formed.

There are two distinct advantages to using LNDS to specify translations. First, NDS, which LNDS extends, has formally defined semantics, which eliminates all ambiguity in translation specification. This makes it possible to check if a translation specification is well-formed and to generate translators from translation specifications. Second, NDS specify the meaning of language features compositionally, which means the features of each language can be understood in isolation.

Chapter 8: Conclusion

In summary, we have presented an approach to compositionally specifying the semantics of non-hierarchical languages. Specifically, we have proposed a method for imposing hierarchical structure onto non-hierarchical syntax using projections. We then were able to specify semantic functions, which operate over projection ASTs, using NDS. As an optimization, we introduced LNDS, which specifies semantic functions that operate over ASGs without transforming them to projection ASTs. To ensure LNDS specifications are well-formed, we have also developed and presented a tool that checks for this property. We view this research into specifying semantic functions as the first step towards generating UML translators.

To that end, we have created Jeanne[1], a prototype for such a translator generator, which generates individually tailored translators for each LNDS specified UML translation. Each generated UML translator translates UML diagrams to target language ASTs, which are pretty printed to produce implementation language code. State of the art UML [7] translation tools, such as Rational Rose [31] and Poseidon [20], allow the developer to choose from different UML to implementation language translations. The different implementation language code generated by these tools are examples of implementation variation. These tools artificially limit implementation variation because the developer is unable to add new translations. For example, database developers who wish to implement their system in SQL are unable use non-extensible UML generation tools that only have C++ specified translations. A UML translation tool is more useful if it provides the desired implementation variation, either by having specified all the translations developers wish to use, which is very unlikely, or by by allowing developers to specify translations. By allowing a developer to spec-

[1]Named after Jeanne Champollion, the mother of Jean Franscois Champollion translator of the Rosetta stone, an allusion to our tool being a generator of translators of the modern hieroglyphic, UML.

ify UML's semantics using LNDS and then use Jeanne to automatically generate a translator that implements this translation, the implementation variation is limitless.

Possible future work includes refining Jeanne. Although we have developed a prototype, there are many nuances of the problem to be explored prior to publication. Upon the completion of Jeanne, we will work to provide fine grain code generation options, granting the developer greater control over the generated code. We are also interested in extending our work by investigating how the use of multiple types of UML diagrams, such as state diagrams, could be used to generate more complete code.

Bibliography

[1] Agrawal A., Levendovszky T., Sprinkle J., Shi F., and Karsai G. Generative programming via graph transformations in the model-driven architecture. In *Proc. of OOPSLA, Workshop on Generative Techniques in the Context of Model Driven Architecture, Seattle, WA, November 5, 2002*, 2002.

[2] Shi F Agrawal A., Karsai G. A uml-based graph transformation approach for implementing domain-specific model transformations. Technical Report ISIS-03-403, Vanderbilt University, November 2003.

[3] Demissie B. Aredo, Issa Traoré, and Ketil Stølen. Towards formalization of UML class structure in PVS. Technical report, University of Oslo, August 1999.

[4] Richard Bird. *Introduction to Functional Programming Using Haskell second edition.* Prentice Hall Europe, 1998.

[5] Michael Blaha and William Premerlani. *Object-Oriented Modeling and Design for Database Applications.* Prentice-Hall, Inc, Upper Saddle River, New Jersey, 1998.

[6] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[7] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[8] R. Bordeau and B. Cheng. A formal semantics for object model diagrams. In *IEEE Transactions on Software Engineering*, pages 21(10):799–821, 1995.

[9] Jean-Michel Bruel. Transforming UML models to formal specifications. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[10] Scott A. DeLoach and Thomas C. Hartrum. A theory-based representation for object-oriented domain models. *IEEE Transactions on Software Engineering*, 26(6), June 2000.

[11] Min Deng. Formal denotational semantics of UML using metamodels and NDS rules for embedded systems domain. Technical report, Michigan State University, 2004.

[12] L. K. Dillon and R. E. K. Stirewalt. Lightweight analysis of operational specifications using inference graphs. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 57–70. IEEE Computer Society Press, 2001.

[13] L. K. Dillon and R. E. K. Stirewalt. Inference graphs: A computational structure supporting generation of customizable and correct analysis components. In *IEEE Transactions of Software Engineering*, 2002.

[14] David Duffy. *Principles of Automated Theorem Proving.* John Wiley and Sons Ltd., West Sussex PO19 IUD, England, 1991.

[15] Martin Erwig. Semantics of visual languages. pages 304–311. 13th IEEE Symposium on Visual Languages, 1997.

[16] J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[17] Andy Evans and Tony Clark. Foundations of the Unified Modeling Language. In *Proc. of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, 23-24 September 1997*, 1997.

[18] Robert France, Andy Evans, and Kevin Lano. The UML as a formal modeling notation. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[20] Gentleware. Poseidon. URL: http://www.gentleware.com/products.

[21] S. Gnesi, D. Latella, and M. Massink. Model checking uml statechart diagrams using jack. In *Proc. of 4th IEEE International Symposium on High-Assurance Systems Engineering*, Washington DC, USA, 1999.

[22] Wai Ming Ho, Jean-Marc Jquel, Alain Le Guennec, and Francois Pennaneac'h. UMLAUT: An extendible UML transformation framework. In *In Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 275–278, 1999.

[23] Paul Hudak, John Peterson, and Joseph Fasel. *A Gentle Introduction To Haskell 98.* 1999.

[24] Soon-Kyeong Kim and David A. Carrington. A formal denotational semantics of UML in Object-Z. *L'OBJET: Software, Databases, Networks*, 7(1), 2001.

[25] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999.* Kluwer, 1999.

[26] Luigi Lavazza, Gabriele Quaroni, and Matteo Venturelli. Combining uml and formal notations for modelling real-time systems. In *Joint 8th European software engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2001.

[27] W. E. McUmber and B. H. C. Cheng. A general framework for formalizing uml with formal languages. In *In Proc. of the 2001 International Conference on Software Engineering (ICSE'2001)*, 2001.

[28] D. Milicev. Automatic model transformations using extended uml object diagrams in modeling environments. *IEEE Trans. Softw. Eng.*, 28(4):413–431, 2002.

[29] D. Muthiayen and V. S. Alagar. Formalizing UML for rigorous software development. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[30] G. Övergaard. A formal approach to relationships in the Unified Modeling Language. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Software Modeling Techniques*, pages 91–108. Technische Universität, München, Germany, TUM-I9803, 1998.

[31] Rational. Rational rose. URL: http://www.rational.com.

[32] Ravi Sethi. *Programming Languages : Concepts & Constructs.* Addison-Wesley, 1996.

[33] M. Shroff and R. France. Towards a formalization of UML class structures in Z. In *In Proceedings of COMPSAC'97*, 1997.

[34] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages.* Addison-Wesley, 1995.

[35] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, New York, 1992.

[36] Kurt Stirewalt and Laura K. Dillon. Generation of visitor components that implement program transformations. In *ACM SIGSOFT Symposium on Software Reusability*, pages 86–94, 2001.

[37] R. E. Kurt Stirewalt and Laura K. Dillon. A component-based approach to building formal analysis tools. In *International Conference on Software Engineering*, pages 167–176, 2001.

[38] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Pub Co., 2000.

[39] Meyer C. Tanuan. Automated analysis of unified modeling language UML specifications. Master's thesis, University of Waterloo, Waterloo, Canada, August 2001.

[40] Enoch Y. Wang and Betty H. C. Cheng. Formalizing and integrating the functional model into object-oriented design. MSU Technical Report MSU-CPS-97-34, Michigan State University, Department of Computer Science, East Lansing, MI 48824, September 1997. Submitted for publication.

[41] Enoch Y. Wang, Heather A. Richter, and Betty H. C. Cheng. Formalizing and integrating the dynamic model within OMT. In *IEEE Proceedings of the 19th International Conference on Software Engineering*, pages 45–55, Boston, MA, May 1997. IEEE.