

DECLARATIVE TECHNIQUES FOR UNPARSING
COMPLEX DATA STRUCTURES

By

Anantha Narayanan

Thesis

Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2004

Nashville, Tennessee

Approved:

Date:

To the memory of my mother,

Shrimati S. Mangalam

ACKNOWLEDGEMENT

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA), Model Based Integration of Embedded Systems (MoBIES) project.

My heartiest thanks go to my advisor Dr Gabor Karsai for his invaluable guidance and support, which kept me motivated throughout the course of this research. I would also like to thank Tamas Paka for his support, without which this project could never be completed.

I would like to thank Dr Sandeep Neema, Aditya Agrawal, Feng Shi, Endre Magyari, and all the others in the MoBIES project for helping me whenever I encountered a problem.

I would like to specially thank Dr Sandeep Neema and Tivadar Szemethy for being the first to test my product and provide valuable feedback.

Last but not the least, I would like to thank my father Suryanarayanan and my sister Anuradhai Mukundan for their encouragement and belief in me. Their support and advice has helped me throughout my academic career. Lastly, I would like to thank Suchismita for her undying affection and unswaying confidence in my abilities.

TABLE OF CONTENTS

	Page
To the memory of my mother,	ii
Shrimati S. Mangalam	ii
ACKNOWLEDGEMENT	iii
TABLE OF CONTENTS.....	iv
LIST OF TABLES	vi
LIST OF FIGURES	vii
LIST OF ABBREVIATIONS.....	viii
Chapter	
INTRODUCTION	2
Existing technologies.....	3
Problems with the existing technologies	4
Requirements of a new technology.....	5
Problem Statement.....	6
Layout of Chapters.....	6
BACKGROUND	8
The Unified Modeling Language.....	8
Universal Data Model.....	10
UDMPat.....	14
The Object Constraint Language	16
Characteristics of OCL	17
Craig Cleaveland's Template Language.....	17
Related Work.....	20
Comparison of Pattern Approaches	21
A NEW PATTERN PROCESSOR FOR UDM	22
Structure of the Pattern file.....	23
The <code>print</code> command	24
The <code>open</code> command	25
The <code>switch</code> command.....	26
Using OCL statements in the pattern file.....	26

Separating plain text and pattern instructions.....	28
Internal Operation of the Pattern Processor.....	28
Generating output from a pattern file	33
EXAMPLES AND EVALUATION.....	36
State Chart code generation example.....	36
State Chart HTML example.....	42
Evaluation	47
CONCLUSIONS AND FUTURE WORK.....	49
Conclusions.....	49
Future Work.....	50
Appendix	
Pattern File for Example 1 (C++ Code Generation from State Chart model)	52
Sample Output generated by the Pattern File Above.....	53
Code for InvokeEx Method for GME Interpreter Equivalent to Pattern File in Example 1	54
Pattern File for Example 2 (HTML Code Generation from State Chart model)	55
Pattern File for Example 2 (continued).....	56
REFERENCES	57

LIST OF TABLES

Table

Page

LIST OF FIGURES

Figure	Page
1. Elements of UML class diagrams	8
2. Usage of the UDM Framework.....	11
3. C++ classes generated by UDM [6]	12
4. Sample OCL classes [2]	16
5. Usage of TL.....	19
6. Usage of pattern processor	22
7. Sample class diagram	27
8. Initial stages of a compiler	29
9. Basic functioning of a lexical analyzer	30
10. Class hierarchy of nodes for the OCL parser tree	31
11. Object diagram showing tree structure of a script statement	32
12. UML Meta model for the State Chart paradigm	37
13. Example model for the State Chart paradigm	38
14. State Chart Model for example 2	43
15. State Chart model for example 2 (Tree view)	43
16. State machine for System 1 in Figure 14	44
17. State machine for System 2 in Figure 14	44
18. State machine for System 3 in Figure 14	44
19. Sample output for example 2	46

LIST OF ABBREVIATIONS

UML – Unified Modeling Language

UDM – Universal Data Model

FSM - Finite State Machine

GME - Generic Modeling Environment

OCL - Object Constraint Language

API - Application Programming Interface

XML - eXtensible Markup Language

DTD – Document Type Definition

BON – Builder Object Network

CHAPTER I

INTRODUCTION

Modern systems are increasingly complex, making their understanding a demanding, yet crucial task. Model-based development of systems is a process that uses explicit domain-specific constructs with well-defined semantics to represent, analyze, and synthesize a system [12]. A model is an abstract representation of an object or a concept. Models can be individual entities representing a physical object, or a network of objects. In several such implementations, converting these object networks into a textual representation is a problem often encountered.

Model based applications today work with large and intricate networks, with thousands of objects and several hierarchical layers. Several tools are currently available for creating and working with such networks of models, such as GME (Generic Modeling Environment) [9], UDM (Universal Data Model) [6], GReaT (Graph Re-writing and Transformation Engine) [14]. Unfortunately, there is usually no easy interface provided for simple unparsing (converting into a flat representation) of such data structures into a textual representation.

Simple text based files such as XML files, configuration files or C/C++ code are often generated from large data networks. However, the APIs provided are not suited for such tasks. Producing text files using *printf* statements is often cumbersome and doesn't give good control over formatting and layout of the resulting text file. They also contain redundancy. In recent applications, as much as 80% of the output has been constant text

that was printed verbatim. Such applications become hard to maintain, with small changes requiring lots of work and recompilation.

There is a need for converting data structures into a textual representation, such that the process is described in a declarative way. It would be convenient to have a simple, pattern based, interpretive method, of inserting data into pre-formatted text files. Such text files must be easy to write and maintain. They must also avoid the necessity to recompile after making small changes to the text file.

Existing technologies

The UDM (Universal Data Model) framework includes the development process and set of supporting tools that are used to generate C++ programmatic interfaces from UML class diagrams of data structures [6]. UDM is used where an object oriented approach is followed for describing data structures. A UDM data network consists of a network of object instances of classes defined first in a class diagram, where attributes and associations can also be included. UDM ships with a pattern processor called UDMPat.

UDMPat takes as input a UDM data network and a specially coded text file and produces a textual output. The text file consists of a mixture of plain text and special instructions. The textual part is printed out verbatim to the output. Special instructions can be included in the text file, which will perform special actions such as traverse through a data network to retrieve data.

The only alternative for generating text output from a data network with UDM was to code using the C++ API provided with UDM. The API offers several methods for retrieving information from the UDM data networks. The methods in the API can be

used to retrieve data, and the output can be generated using *printf* statements, to print plain text or data from models.

Problems with the existing technologies

UDMPat, though it is very simple and quite powerful, has not been widely used because of several drawbacks. It uses special commands which the user must learn before using the utility. This takes extra effort, and is not helpful to the user beyond the scope of UDM and UDMPat. Though the text is printed out verbatim and can be formatted as the user likes, the syntax of UDMPat offers very little formatting of data.

UDMPat offers very limited control over traversing of the network, and it is not expressive enough. For instance, it does not allow conditional selection, or selection of objects of multiple types. There are no simple methods to count the number of elements in a set, or iterate through them in a particular order. It offers poor conditional constructs, and simple comparisons such as string comparisons are difficult to achieve. A more powerful way to access and handle data is required.

The failure of UDMPat led most programmers to using the C++ API. Though powerful in its functionality, it is difficult to use. Generating a simple text output file may involve several *printf* statements, several escape sequences and repetitive code. The format of the output cannot be specified easily. Plain text which will be constant still has to be printed through *cout* or *printf*. This makes the code difficult to read and the output difficult to predict.

Another important problem with this approach is that the code must be compiled for every small change. Coupled with the fact that the output is not easily predictable, this

makes generating a desirable output very difficult. It also means that maintaining these for later changes will prove costly.

Requirements of a new technology

Since simple textual files are being generated increasingly often, there is need for a new technology that combines the good features of the existing technologies, and offers users with a powerful and easy to use tool to generate a textual representation from a large data network. A simple pattern processor that processes a pre-formatted text file interspersed with special instructions to insert data seems the right way ahead.

The features required of a good pattern processing tool seem to be the following [7]:

- It must offer a concise and powerful expression language for accessing data
- It must provide a concise way to insert data into the output
- It must have concise and powerful iterative and conditional constructs
- It must allow powerful and easy control over white-space and the general formatting of the output text.

Considering these requirements, this thesis looks at a new tool for processing a data-network to generate a textual output. OCL has been chosen as the language for traversing the data, as it is tightly coupled with UML and provides a vast range of functionality. OCL offers excellent iterative and conditional constructs, and a wide range of data types that help in handling various kinds of objects including collections.

Problem Statement

My objective is to develop a pattern processing tool, which can take an input text file and produce an output text file (or several output text files) from an object network by parsing special scripts embedded in the input text file. The inputs provided will be the following:

Data locator: The file containing a data network. For the three persistence technologies supported by UDM, this is usually an MGA, XML or MEM file. This will be the source of data for the output

UML diagram: The file describing the meta-model for the above data. This file can be generated by the UML interpreter in GME (Generic Modeling Environment) [9].

Pattern file: This is a text file describing the desired output. It contains plain text and scripts, and must be written by the user according to the output he/she wants to produce.

The pattern processor will send the plain text verbatim to the output. It will parse the scripts to retrieve data from the data network and insert it as specified into the output. The pattern processor must be independent of the meta-model and must be usable with any paradigm and with any data network. Pattern files written for models based on a certain meta-model should work with any data network based on that meta-model.

Layout of Chapters

Chapter 2 follows with an introduction to the relevant standards and tools prevalent today. It will discuss the basics of UML, UDM and other technologies. Chapter 3 describes the construction of the pattern processor and its usage. It will discuss the choice of scripting language, notation etc. and the motivation behind these choices.

Chapter 4 will bring forward some examples of using the pattern processor. It will finish by comparing the task of writing a pattern file with writing C++ code to achieve the same result. Chapter 5 lists the conclusions drawn and the scope for future work in this direction.

CHAPTER II

BACKGROUND

The Unified Modeling Language

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [3]. UML is an OMG (Object Management Group [4]) standard that defines the basic artifacts and rules for constructing models in a standard language.

UML defines the modeling elements (fundamental modeling concepts and semantics), notation (visual rendering of model elements) and guidelines (idioms of usage within the trade) for building object-oriented, component based systems [3].

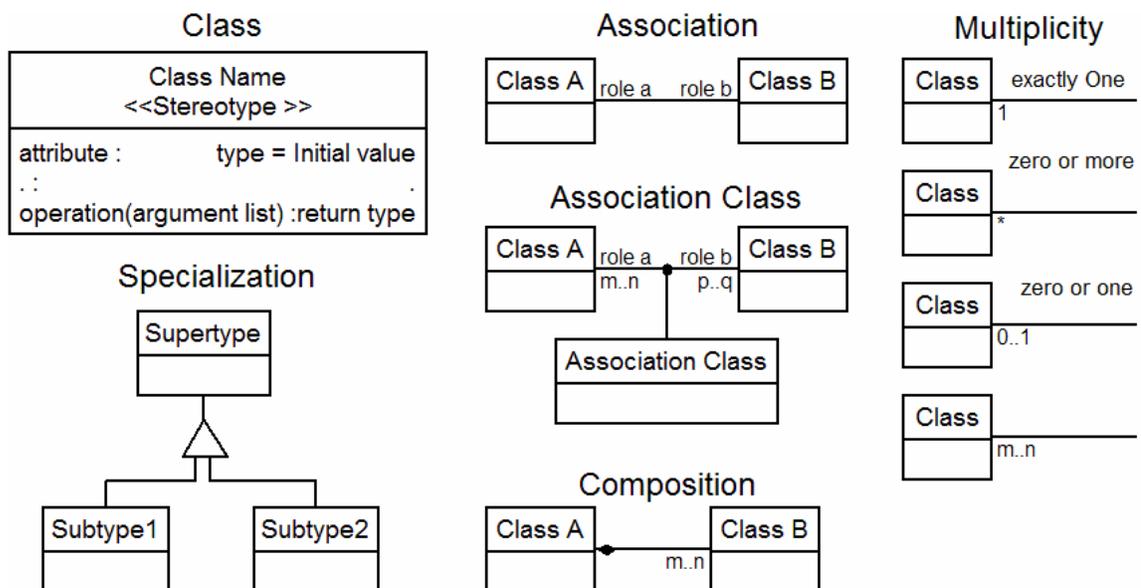


Figure 1: Elements of UML class diagrams

The most commonly used artifact of UML, and the one we are primarily concerned with, is the UML class diagram. The UML specification [3] defines the notation and semantics to represent objects as classes with attributes and operations, and associations between such objects such as inheritance and aggregation.

UML class diagrams are a convenient and unambiguous way to represent several aspects of complex systems. Figure 1 shows the basic elements of a UML class diagram. A class is represented as a rectangular box containing the class name, and its stereotype enclosed in angular brackets (guillemots). It is followed by the list of attributes of the class, along with the type of each. This is in turn followed by the list of operations that can be performed on the class, and the return types of those operations.

Associations between classes are represented by lines connecting the classes, which may in turn contain special symbols indicating the nature of the association. Inheritance is represented by a triangle. In the figure shown, the classes “subtype1” and “subtype2” are inherited from the class “supertype”. This means that “subtype1” and “subtype2” have all the attributes and operations of “supertype”. Composition is a special kind of association that represents a containment relationship, and is represented by a shaded diamond. The placement of this diamond is important, as it defines which class contains which. In the figure shown, “Class A” contains “Class B”. The cardinality, i.e. the number of instances that can be contained, is also represented on the association line. Another type of association is aggregation, which represents a part/whole relationship. The difference between aggregation and composition lies in the lifetimes of the contained class. In a composition, the contained class ceases to exist when the containing class ceases to exist. In an aggregation, the ‘part’ class can exist even without the ‘whole’

class. Aggregations are however not directly supported in UDM. All other associations are represented by a simple line, with the role of each class in that association and the cardinalities shown along with the association.

Universal Data Model

The UDM includes the development process and set of supporting tools that are used to generate C++ programmatic interfaces from UML class diagrams of data structures. These interfaces and the underlying libraries provide convenient programmatic access and automatically configured persistence services for data structures as described in the input UML diagram [6].

The storage technologies currently supported are as follows:

- XML with an automatically generated DTD file,
- MGA, the native interface of the GME modeling environment
- Memory-based storage.

UDM is typically used in any object-oriented approach where objects are first defined in a UML class diagram. These can then be instantiated programmatically. UDM has the tools to generate a convenient C++ API from the data structure in the UML class diagram. This API gives access to all the components of the class diagram, and all the methods necessary for building, navigating and manipulating a network based on the class diagram. UDM ships with the UDM generator, a UML paradigm for GME (which allows users to draw UML class diagrams using GME), an interpreter for this paradigm (which generates an XML file from the UML class diagrams) and a number of supporting applications.

Figure 2 describes the common usage of UDM. We start with a UML class diagram created in GME. An XML file is generated from this class diagram, using the provided interpreter. When the UDM generator is executed, passing this XML file, it generates some header (.h) and program (.cpp) files. These can be used in any user application, along with the other UDM libraries, providing a convenient API for the user to access components of a data structure based on the initial class diagram. The UDM application also generates a DTD file for accessing data networks defined in XML. The UDM application also generates a DTD file for accessing data networks defined in XML.

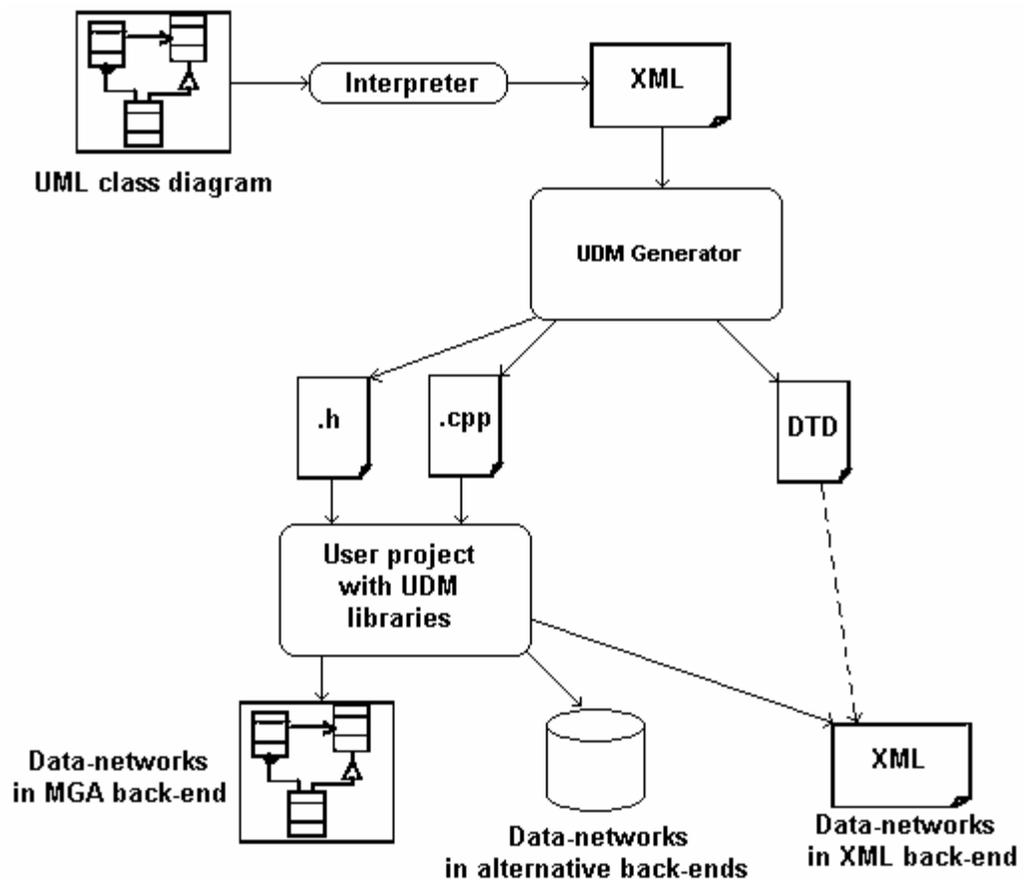


Figure 2: Usage of the UDM Framework

For each UML class in the source diagram, UDM generates a C++ class with the corresponding name. All classes in UDM are derived from the base class `Udm::Object`, which defines the generic functionalities of objects in UDM. Factory methods are provided to create class objects for these classes. Inheritance in the UML class diagram is reflected as C++ public inheritance. Figure 3 shows a simple mapping of UML classes to UDM.

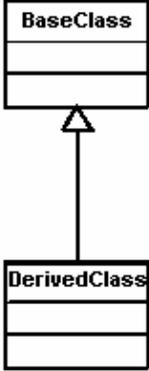
 <pre> classDiagram class BaseClass class DerivedClass BaseClass < -- DerivedClass </pre>	<pre> namespace ExampleDiagram { Udm::Object Object; ... class BaseClass : public Object { ... }; class BaseClass : public Object { ... }; } </pre>	<pre> // assignment to same type int f1(DerivedClass &c) { DerivedClass c2 = c; } //from derived to base int f2(Baseclass &b, DerivedClass c) { b = c; } //from base to derived int f3(DerivedClass &c, BaseClass &b) { //check if compatible if(b.meta() == DerivedClass::meta) { c = b; } } </pre>
Class Diagram	UDM Header	User Code example

Figure 3: C++ classes generated by UDM [6]

For each class in the UML diagram, a C++ class with the corresponding name is defined by UDM. All classes belong to a namespace with the same name as the UML class diagram. Variables exist as instance objects in the back-end, and handles are provided for setting the variables. For each UML attribute in a class, access methods are defined in the corresponding C++ class. UML diagrams may contain access permissions on attributes, but these are not generated in the interface, where all attributes are

considered public. Inheritance is reflected as C++ public inheritance, thus giving seamless access to attributes and methods of the base class from the derived class.

UML composition relationships are reflected as access methods in both the containing and contained classes, which return instances of wrapper classes. All objects in UDM must be contained in exactly one parent. **Udm:ParentAttr<childObject>** represents the single parent of the object. It can be assigned to any object of type *childObject*. **Udm::ChildAttr<parentObject>** is used to represent a contained object when the maximum multiplicity on the child side is 1. It can be assigned to any object of type *parentObject*. When the maximum multiplicity on the child side is greater than 1, **Udm::ChildrenAttr<parentObject>** is used, which can be assigned to any object of type *set<parentObject>*. New relationships can be assigned in this way.

Similarly, access methods are generated on the parent and child side for accessing the parent and children of the objects.

UDM data networks are organized in a single tree, i.e. each object (except the root object) is contained in exactly one parent. New objects are created using the static `Create` method of each object, passing the parent object. The code below shows the `Create` method in UDM generated classes:

```
class A : public Object {
...
    static A Create(const Object &parent,
                    const CompositionChildRole &role = Udm::NULLCHILDROLE
    );
...
}
```

UDMPat

UDMPat is one of the generic programs packaged with UDM. UDMPat reads UDM data while processing a pattern script to generate textual output to one or several files. The pattern file consists of a mixture of plain text, and special pattern instructions. The plain text is sent verbatim to the output. The special instructions are evaluated by the pattern processor, and the result is sent to the output [6].

Since this is a generic program, the data structure information is supplied at runtime. UDMPat takes three arguments:

<in-data>: The input UDM data network

<diagram>: This is an XML file that represents the meta diagram (the UML class diagram based on which the data-network is created).

<pattern-file>: the name of the pattern script

The pattern script can contain several types of instructions for performing several tasks. The most relevant instructions are the ones that retrieve information from the input UDM data. Using these, it is possible to access attributes, and iterate through associations and containment relationships.

UDMPat always accesses the UDM data in read-only mode. The pattern script is always executed in the context of a UDM object. Some instructions can be used to change the context. The most commonly used instructions, and their main drawbacks are listed below.

The `$<variable-name>` is the simplest command, used to return the value of a variable. This variable may be defined earlier using the `$DEFINE(<variable>, <value>)` command, or an attribute of the object currently in context. While these commands allow

defining and retrieving simple variables, they do not allow simple operations on the variables such as string concatenation or ordering of a set.

The `!EVAL_FORALL(<fieldspec>, <arg>)` command gets the set of objects identified by `<fieldspec>` and iterates through them. This is the only iterative construct provided, and is very limited. It does not allow the user to select a set of objects based on a special condition such as the value of a certain attribute. The order of the iteration can not be controlled by the user. Nested iterations with arbitrary sets of objects is very difficult to achieve.

The `!IFEMPTY(<arg1>, <arg2>)` command evaluates `<arg2>` if `<arg1>` evaluates to an empty string. This is the only conditional construct. It is not intuitive, and simple numerical and string comparisons cannot be achieved easily.

The `!TO_FILE(<arg>)` command directs output to the file specified by `<arg>`. It first closes the current file, before opening a new file. This makes switching between outputs impossible. Once a file is closed, it cannot be reopened to append data. Reopening the file causes all its existing data to be erased. Existing files are always overwritten.

The `!POSTINCR(<varname>)` command returns the value of `<varname>`, and increments its value by 1. This is the only way to increment numbers. Simple mathematical operations such as addition or multiplication cannot be performed.

These are the main drawbacks with UDMPat, which has led to a search for a pattern language that will offer more control, flexibility and ease of use to the user.

The Object Constraint Language

The Object Constraint Language (OCL) is another OMG standard for building software models. It is defined as a standard “add on” to the UML [2]. Expressions written in OCL add vital information to the models written using UML.

A diagram cannot express all the statements that must be a part of a thorough specification. For instance, consider the UML diagram shown below. The association *passengers* associates a flight to a number of instances of the Person class. But the number of instances is limited by the *numberOfSeats* attribute of the Airplane class. There is no way in the UML diagram to specify this restriction.

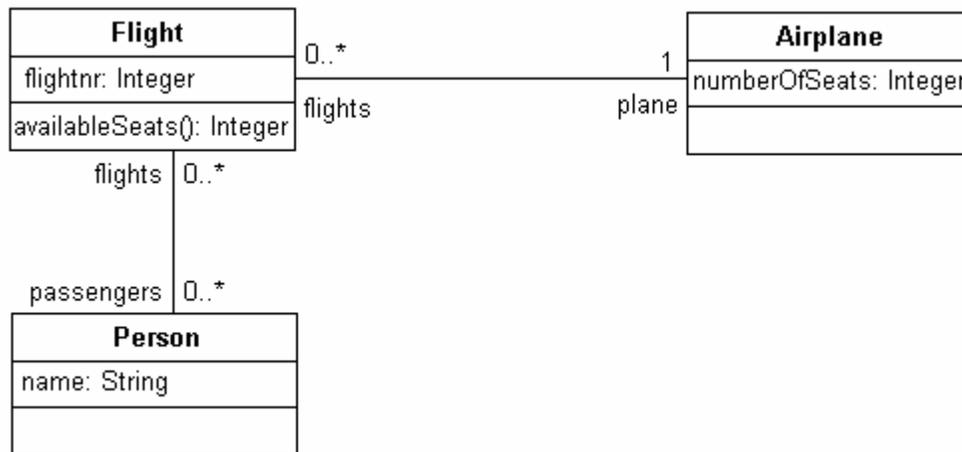


Figure 4: Sample OCL classes [2]

This restriction can be overcome by using a OCL constraint:

```
Context Flight
```

```
Inv: passengers->size() <= plane.numberOfSeats
```

In this way, expressions written in a precise, mathematically based language like OCL offer a number of benefits over the use of diagrams to specify a system. However, a

system specified in a language that uses an expression representation alone is not easily understood. A combination of UML and OCL offers greater power in specifying models.

Characteristics of OCL

OCL is a constraint definition language but can also be used as a query language. A constraint is defined as follows: “A constraint is a restriction on one or more values of an object-oriented model or system.” [2]. Because OCL expressions can indicate any value or collection of values in a system, OCL has the capabilities similar to SQL. The body of a query expression can be completely specified by a single OCL expression.

OCL is based on mathematical set theory and predicate logic, and has formal semantics. The notation, however, does not use any mathematical symbols. The result is an unambiguous language that is easily read and written by all [2].

OCL is a strongly typed language, such as Java, C# etc. It is also a declarative language. OCL expressions simply state relationships between variables in terms of functions or rules. The OCL parser can apply some algorithms on these relations to produce a result. We are interested in processing such expressions to retrieve and print data into a text file.

Craig Cleaveland’s Template Language

Craig Cleaveland, in his book titled *Program Generators with Java and XML* [7], creates a language for writing templates for program generators whose input uses XML. In this book, a pattern language called Template Language (TL) is developed, for creating program generators in Java. TL can be written in two forms – a simple form

using the '#' character as the delimiter, and a somewhat verbose XML form. The figure below describes the usage of TL.

The initial template can be written in its simple or XML form. A translator is provided to easily convert between the two forms. From this template, the program generator is generated using TL2Java. When this program generator is compiled and executed, providing the XML input file, it generates the required program based on the template and the provided inputs.

TL incorporates the idea of retrieving data from a network (in this case, an XML file), and producing a text file output. While it is mainly designed to generate Java programs, it can generally be used to generate any kind of text file.

An example template file in simple form is given below:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello #">//name"#");
    }
}
```

The same template in XML form is as below:

```
<?xml version="1.0"?>
<t1>public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello <value path="//name"/>");
    }
}</t1>
```

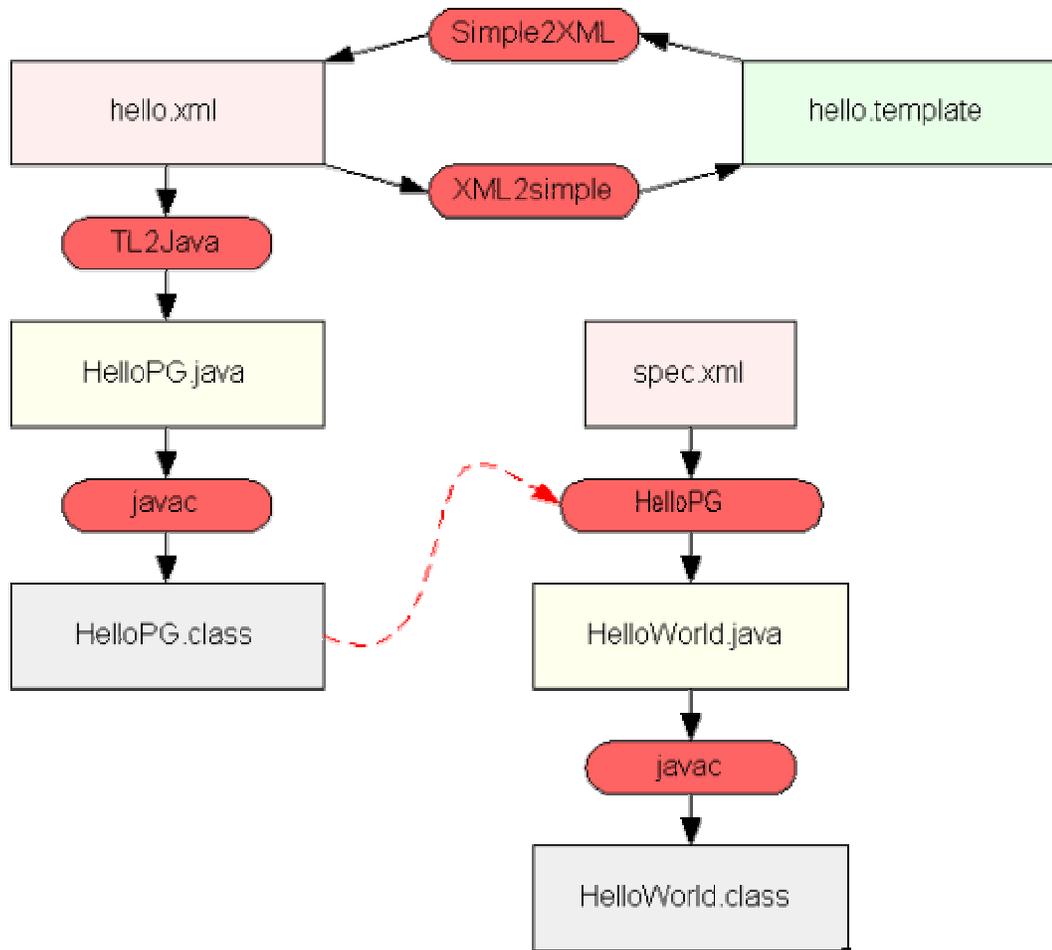


Figure 5: Usage of TL

The TL2Java tool parses this template and generates a program generator. The program generator is a Java program that produces a text output based on the program template and an XML file containing some data. The plain text in the template is printed out verbatim by the program generator. It then uses the Java XML parser to evaluate the special XPath expressions in the template. When the program generator is executed, it queries the provided XML and inserts data from the XML in the specified locations. The above template will produce the following output for the given XML file:

XML input for the template:

```
<?xml version="1.0"?>
<name>Craig</name>
```

Output produced:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello Craig");
    }
}
```

Related Work

The Gandalf project [15] [16] is concerned with semi-automatically generating software development environments that integrate programming and system development environments. It uses a syntax directed editor to provide a language oriented environment. Such editors maintain programs internally as a syntax tree, which is unparsed to display text on screen [17]. Since the syntax tree provides a uniform internal representation, such editors can replace text editors and parsers. The templates for such syntax trees can be related to meta-models, and the resulting programs governed by these templates can be related to object networks based on the meta-model.

In [18] tree-structured data is compared to relational databases, in an attempt to describe a query language for such data, which will be as easy to use and as expressive as SQL. The attention is directed to pattern languages based on regular expressions.

DiSTiL [19] is a software generator implementing a declarative domain specific language for container data structures. The paper stresses the importance of a declarative language that is specific to the domain of the data structures. DiSTiL extends the C programming language with declarative statements and operations on data structures. We

will see later that OCL is a suitable declarative language for querying data structures, and is domain specific by nature.

Comparison of Pattern Approaches

The above sections described some approaches for generating text files from data networks. The basic idea revolves around specifying plain text and special instructions in a text based input file, which when processed by an interpreter, will generate a text output file containing the plain text and data extracted from the data network.

UDMPat uses its own pattern language for querying data from the data network. It has primitive iterative and conditional constructs. The pattern language is not very expressive and not easy to understand.

TL uses XPath as the querying language. It first generates an intermediate Java program called the “program generator”. This must be compiled and executed to obtain the final output. This intermediate step is often cumbersome, and it is preferable to have a tool that will generate the output in one pass, without a need for compiling.

The next chapter describes the new pattern processor developed in this thesis, and the choice of a pattern language for the processor.

CHAPTER III

A NEW PATTERN PROCESSOR FOR UDM

A new pattern processor is developed for UDM in this thesis. This application is capable of generating a text output given any UDM data network and a pattern file. The pattern file can contain plain text interspersed with special instructions that will be described in this chapter.

The figure below shows the typical usage of the pattern processor. It takes three inputs, and processes them to produce a text file output. The first input is the data network to be parsed. This can be in any of the back ends provided by UDM (XML, MGA or MEM). The second input is the diagram file used to generate the data. The final input is the pattern file which describes the output.

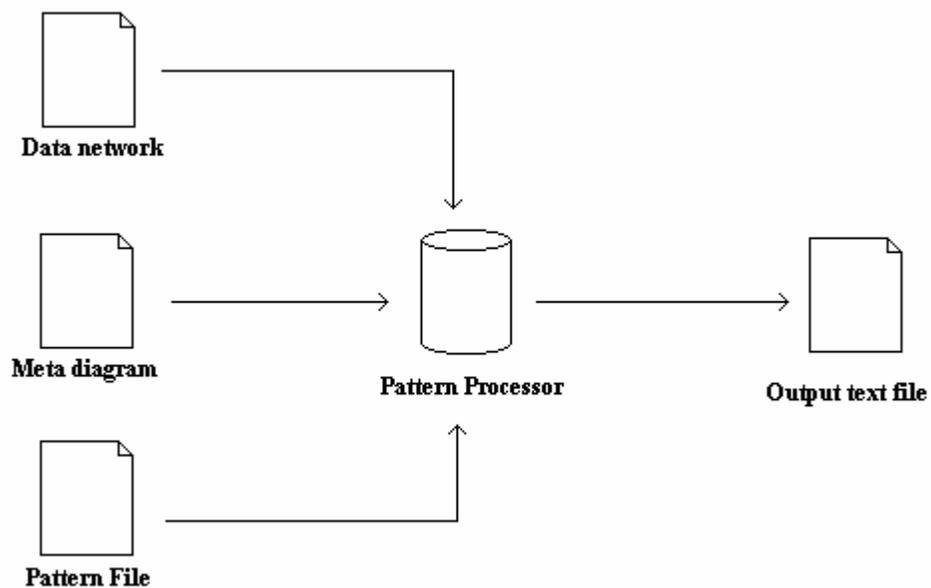


Figure 6: Usage of pattern processor

The pattern processor is available as a command line application, or as a library function. The command line application is invoked in the following manner:

```
UdmOclPat <data locator> <diagram> <pattern file>
```

The first parameter is the data network file. The second parameter is the meta-diagram. The meta-diagram represents the UML class diagram from which the data-network is designed. This is usually the XML file generated from the GME UML meta-model for the data by the GME UML Interpreter. The third parameter is the pattern file. In the command line application, the context for evaluating pattern instructions is always the root object of the data network.

The library function is invoked in the following manner:

```
UdmPat::ProcessPat(<data network>, <context>, <pattern string>);
```

This function is in the UdmOcl.lib library. The first parameter is the UDM data network object containing the data network to be parsed. The second parameter specifies an object in the data network that will be the context for the expressions in the pattern file. All OCL expressions in the pattern file will be evaluated in this context. If the data network is *nw*, this parameter can usually be set to *nw.GetRootObject()*, which returns the root object of the data network. When using the library function, it is not necessary to explicitly pass the meta-diagram.

Structure of the Pattern file

The pattern file provides a powerful, convenient and flexible way to describe the recipe to generate a text output from a UDM data network. The pattern file consists of two parts – plain text that is printed out verbatim, and special instructions that are

processed by the pattern interpreter, and the result sent to the output. The special instructions in the pattern file are separated by the “<:” and “: >” tags. These tags may enclose OCL expressions or any of the additional special instructions that will be discussed below.

The most basic form of the pattern file is a simple text file without any special instructions. Such a text file will just be copied to the output, irrespective of the data network it may be used with. For instance, the following pattern is just plain text and will be printed as it is:

```
Hello World.
```

This can be enhanced by adding a plethora of special instructions to obtain any desired output. For instance, we might want to replace “World” with some data from the data network.

The print command

The print command is the most basic of the special instructions. It can be used to print any data to the output. The syntax for the print command is:

```
print( <expression> );
```

where <expression> is any OCL literal or expression. OCL literals are strings, numbers or Boolean literals such as “true” or “false”. The expression enclosed in parenthesis is evaluated, and the result is printed to the output. If the expression results in any of the basic types in OCL, namely string, real, integer or Boolean, its value is printed. In some cases, an OCL expression may evaluate to an OCL object such as a class. If it results in an OCL object, it cannot be printed. Currently, [undefined] is printed in this case.

Now we can modify the pattern to print the name of the root object in the data network, in the following way:

```
Hello <: print( self.name ); :>.
```

In this case, “Hello ” is plain text, and is printed as is. The “<:” tag indicates the beginning of special instructions. In this case, there is only one instruction, namely, `print`. The argument specified for `print` is `self.name`. `self` refers to the context object. In the case of the command line application, it will be the root object of the data network. “`self.name`” will result in the value of the “name” attribute of the root object (or an error, if the root object has no such attribute). This value is retrieved from the data network, and sent to the output by the `print` command. The closing “: >” tag indicates the end of special instructions. The remaining text, including the “.” character, is copied to the output as plain text.

The open command

The output produced by the pattern processor is sent to standard output by default. The `open` command allows the user to direct this output to a file. The syntax for the command is as follows:

```
open(<file name>, <mode>, <handle>);
```

The first parameter specifies the name of the file to which output must be sent. This can be a simple string or an OCL expression, giving the user the option of deciding the file name at run time. For instance, the name can be based on some attribute of some model in the data-network, such as the name of a class. The second parameter is a string specifying the mode in which the file is opened. There are three possible modes: *overwrite*, indicated by “o”, *append*, indicated by “a” or *fail*, indicated by “f”. When a

file is opened in overwrite mode, any existing file with the same name is overwritten. When opened in append mode, the output produced is appended to the file if a file of the same name already exists. In fail mode, the processor terminates with an error if a file with the same name already exists. The final parameter is the handle, which is a simple identifier that can be used subsequently to access the opened file.

The `open` command by itself does not direct output to the file immediately. It merely opens a file for output and assigns a handle to it. It must be used with the `switch` command that is described next.

The `switch` command

A pattern file can contain multiple `open` commands, giving the user the ability to open multiple files for output. The pattern processor identifies the output stream by a handle, and sends all the output to that handle. The `switch` command decides which of these handles the output is sent to. The syntax for the `switch` command is:

```
switch( <handle> );
```

The only parameter is the handle, which must be defined previously in an `open` command.

When used properly together, the `open` and `switch` commands give the user the ability to open multiple files and switch output between files easily.

Using OCL statements in the pattern file

The three basic commands for producing output were discussed above. In addition to these, the pattern file can contain a number of statements in OCL that can be used to navigate in the data network.

Since OCL is being used as a scripting language here, some restrictions have been added to create a suitable syntax and structure. All OCL statements must end with a semi-colon. The body of any iterator must be enclosed within curly braces. Also, the OCL statements must fall within the “<:” and “:>” tags to be processed by the pattern processor. With these restrictions, any OCL statement can be used in the pattern file. A simple example below shows the usage of an OCL statement in the pattern file.

Consider the class diagram below. It describes a meta-model for the example that follows.

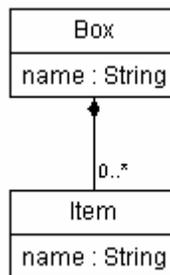


Figure 7: Sample class diagram

The diagram describes a class Box that can contain any number of “Item” instances. For a data network based on this, the following pattern instruction will list the name of the Item instances contained in a Box:

```
<:    self.Item->forAll( i | { print(i.name); });    :>
```

In the above statement, “self” points to the current context. It is assumed here that the context is an instance of Box. The detailed examples that will follow later will provide more detailed explanation of a variety of OCL statements and their usage in the pattern file.

Separating plain text and pattern instructions

All text in the pattern file that is not enclosed within the “<: :>” tags is considered plain text, and will be sent to the output verbatim. The processor will attempt to process any text enclosed with these tags as a special instruction. It is important to understand the usage of these tags to achieve the desired output. Plain text can be embedded within pattern instructions by closing and opening these tags appropriately. For instance, the following instruction adds to the pattern instruction described above to produce a modified output:

```
<:    self.Item->forAll( i | { :>
        The Box contains a <: print(i.name); :> \n <:
    });
:>
```

This will produce a modified output by adding the text “The Box contains “ in front of the Item names. The “\n” produces a new line at the end. Alternatively, the same output can be achieved using the following script:

```
<:    self.Item->forAll( i | {
        print("The Box contains a " + i.name + "\n");
    });
:>
```

This describes a typical result of embedding plain text within special instructions. The more detailed examples that follow will delve deeper into mixing plain text and pattern instructions.

Internal Operation of the Pattern Processor

The pattern processor uses an ANTLR [11] generated parser to parse the input pattern file. A tree structure is created from the pattern file, and it is traversed to perform

the functions specified in the pattern script and produce the required output. The existing OCL parser for UDM was extended to allow for the processing of the special instructions described above.

ANTLR, ANOther Tool for Language Recognition, (formerly PCCTS) is a language tool that provides a framework for constructing recognizers, parsers, compilers, and translators from grammatical descriptions containing Java, C#, or C++ actions [11]. An ANTLR grammar file consists of a list of constructs specifying grammar elements. Action code can be attached to each element, specifying what action must be taken when that element is encountered. Parsers fall into two main categories, the top-down parsers and the bottom-up parsers, based on the order in which the parse tree is generated [8]. Recursive-descent parsing is a top-down parsing method, in which a set of recursive procedures are executed to parse the input. ANTLR generates a recursive-descent parser.

The ANTLR grammar for the OCL parser for GME lists OCL elements such as the various expressions and literals. Action code is included to create a tree structure from these elements. This grammar was extended, to create nodes for the special structures encountered in a pattern file.

The initial stages of a compiler, such as the one generated by ANTLR, are shown in Figure 8 below:

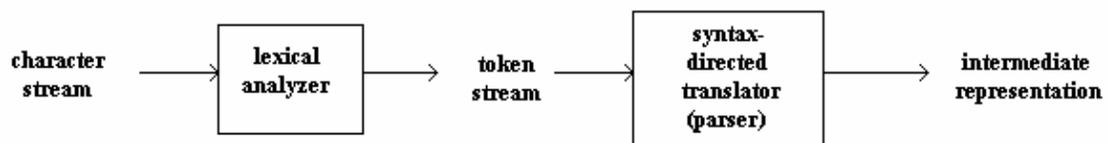


Figure 8: Initial stages of a compiler

The grammar lists groups of characters forming tokens, which will be recognized by the lexical analyzer. The lexical analyzer reads the input file character by character, and returns a string of tokens. Some of the functions performed by the lexical analyzer are recognizing literals, identifiers, keywords etc., and handling white space in the input file. In some cases, the lexical analyzer has to read some characters ahead before it can predict a token. This is called “look-ahead”. If the look ahead does not result in a new token, the extra character has to be pushed back to the input. The output of the lexical analyzer is sent to the syntactical analyzer or parser. Figure 9 below shows the functioning of the lexical analyzer.

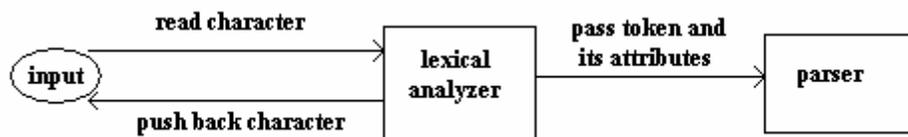


Figure 9: Basic functioning of a lexical analyzer

The parser obtains a string of tokens from the lexical analyzer, and verifies that the string can be generated by the grammar for the source language. The ANTLR grammar for the OCL parser contains a list of such valid strings of tokens. ANTLR allows the addition of action code along with such declarations, which will be executed when a particular string of tokens is encountered. Action code was added in the ANTLR grammar for the OCL parser to generate a tree representation from the input OCL statements. The tree representation consists of several C++ classes. A basic class `TreeNode` is defined, which can be part of such a tree. Various classes are defined for representing the various types of nodes, which inherit from the basic class `TreeNode`.

Each node in the tree implements certain methods that assist in the traversal of the tree. The nodes implement a method called *checkImplementation*, which checks the syntax of the OCL expressions. It is necessary to perform this check before attempting to evaluate the OCL expressions. A method called *Evaluate* evaluates the expression at that node. The Evaluate method is also used by the pattern processor to execute the special instructions in the pattern file.

There are two basic nodes in the pattern processor, called *TextNode* and *EnumerationNode*. The part of the pattern file that is printed out verbatim forms the text nodes. The text node has a string attribute which holds the text to be printed. All script sections are added to the tree as enumeration nodes. These enumeration nodes can then contain more enumeration nodes, text nodes, or a varied number of nodes depending on the actual script command. For instance, a *PrintNode* represents a print command, and contains a *TreeNode* which represents the OCL expression which is to be evaluated and printed. Figure shows some of the main nodes that form the tree and their relationships in a class diagram. The OCLParser has several nodes such as *IteratorNode*, *IfThenElseNode*, *FunctionNode* etc., which are not shown in the figure.

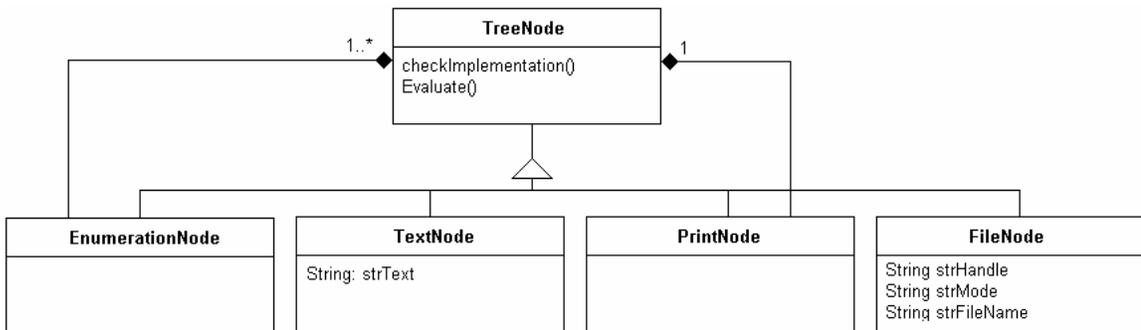


Figure 10: Class hierarchy of nodes for the OCL parser tree

The top level in the tree is always an *EnumerationNode*. The pattern processor adds “<: {” and “} :>” tags to the pattern file, converting the entire pattern file into one enumerated expression. The subsequent text and instructions in the pattern file are added as children to this top level node.

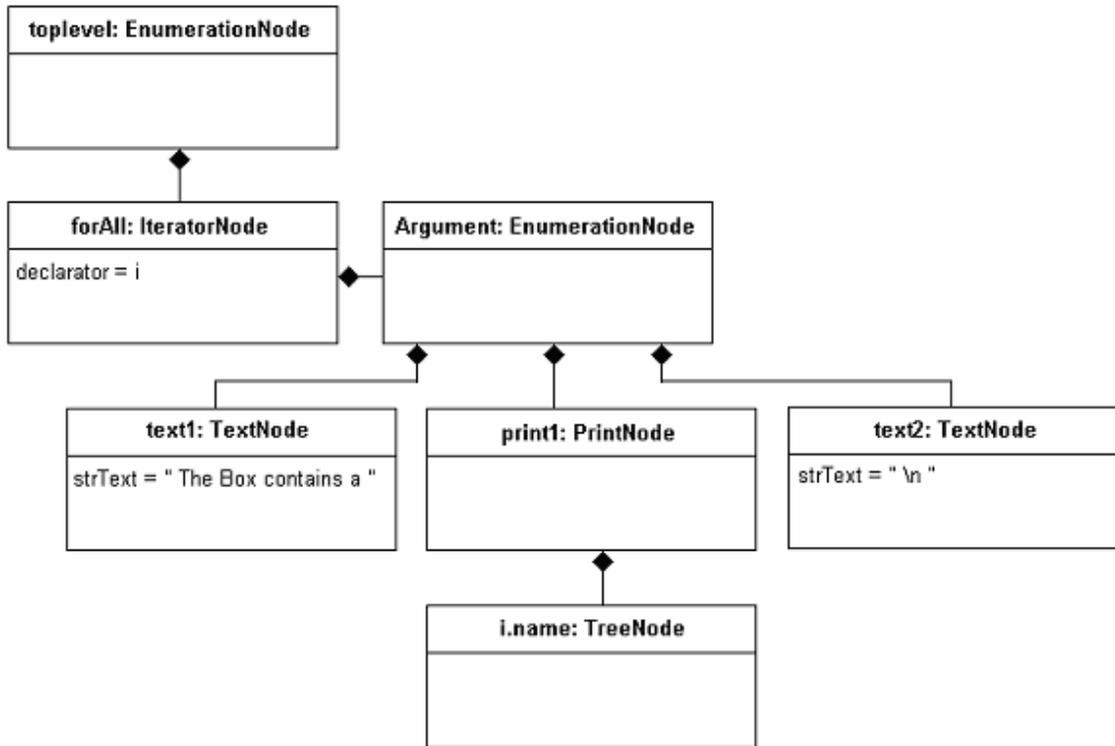


Figure 11: Object diagram showing tree structure of a script statement

Let us look at how a tree is generated from a simple pattern file. For our example, consider a pattern file with the following text we have seen before:

```

<: self.Item->forAll( i | { :> The Box contains a <:
print(i.name); :> \n <: } ); :>
  
```

The top level enumeration node contains one enumeration node for the `forAll` OCL statement. The argument of the `forAll` statement contains a mixture of plain text

and special commands. It thus contains a *TextNode*, followed by a *PrintNode*, followed by the final *TextNode*. Figure 11 depicts the resulting tree.

Notice that the *PrintNode* contained in the `forAll` statement is shown to contain a *TreeNode*. All *TreeNodes* can be broken down to the individual command nodes representing the OCL statement, expression or literal. The pattern processor uses the OCL parser that is shipped with GME and UDM. The OCL parser builds a tree for each OCL statement, which becomes a part of the larger tree in the pattern processor.

Generating output from a pattern file

Once the tree structure has been built and verified, the Evaluate method is called on the nodes to produce the output. In the OCL parser for GME, the Evaluate method parses the expression and returns a Boolean result. This suffices in the GME environment, as OCL is used solely to define and check constraints on models. But in the pattern processor, we are more interested in getting information from models and printing this information on a persistent media. The Evaluate method was modified appropriately to achieve this effect.

Most of the functionality provided by the pattern processor through the OCL parser are a result of “side effects” of executing an OCL statement. For instance, the OCL statement `system.state->forAll(name <> "init")` returns a Boolean result, which is true if none of the “state” objects in a “system” have the attribute “name” set to “init”. To parse this statement, the OCL parser passes through the collection of all “state” objects in a “system” object, and checks the condition for each instance. The conjunction of the results of the comparison for all the instances is then returned as the result of this statement.

We use the side effect that the parser evaluates the expression within the parentheses for each instance of the object in the collection. Thus, the statement `system.state->forall({ print(name); });` in the pattern file will cause the parser to execute the print command on every instance in the collection. Notice that the syntax of the statement in the pattern file differs slightly from that of a normal OCL statement, containing semicolons and curly braces.

One important point must be noted here. In the `forall` example above, it suffices to stop evaluating the expression when the first negative instance is encountered. Once the parser encounters an instance where the condition fails, it can safely return “false” without checking the remaining instances. But this is not desired for the pattern processor. However, the OCL parser can be configured to always evaluate the expression for every instance irrespective of the result of any particular instance. For the pattern processor, the OCL parser has been configured in this way.

The *Evaluate* method is the key to evaluating any statement by the OCL parser. The implementation of this method for the new nodes added for the pattern processor is discussed below.

The `EnumerationNode` simply contains a list of nodes. Its *Evaluate* method simply calls the *Evaluate* method of the contained nodes sequentially. Where relevant, it returns the value returned by the *Evaluate* method of the last of the nodes contained in the `EnumerationNode`. The significance of this return value is discussed later with an example in the next section.

The `PrintNode` contains an OCL expression, the result of which must be printed. The *Evaluate* method for this node calls the *Evaluate* method for the node for the

expression it contains. It then attempts to print the result returned into the current stream. Strings and numbers are printed as they are, and `[undefined]` is printed where the result is an OCL object.

The `TextNode` simply contains plain text as a string that must be printed verbatim. The `Evaluate` method for this node prints the string to the currently selected output stream.

CHAPTER IV

EXAMPLES AND EVALUATION

This chapter explains the usage and functioning of the pattern processor using some examples. A step-by-step approach is taken to explain the construction of a pattern file to achieve a desired output. The complete pattern files in the examples here can be found in the Appendix.

State Chart code generation example

For the first example, we will look at generating C++ code that will implement a state chart as defined in the state chart paradigm for GME shown in figure 1. The state chart paradigm for GME is the equivalent of a UML class diagram that defines classes for states, compound states such as OrState and AndState, and transitions between these states. The transitions have attributes for setting triggers, guards and actions. A sample state chart model is shown in Figure 13, containing some state atoms and transitions between them.

The goal is to generate a C program that will implement any state machine defined using this paradigm. A generic tool is required that can take any model based on the paradigm, parse the data network and produce a text file that will contain C code appropriate for that model. The pattern processor fits the description perfectly.

The first step is to design a sample output, which will be the template for the more generic pattern file. For this example, we will use two files. The first will be a C program called *state.c* that will allow the user to enter a value specifying the trigger, which might

cause the state machine to transition to a different state. The program will then find the resulting state from the trigger and its current state, if there is a valid transition for that trigger.

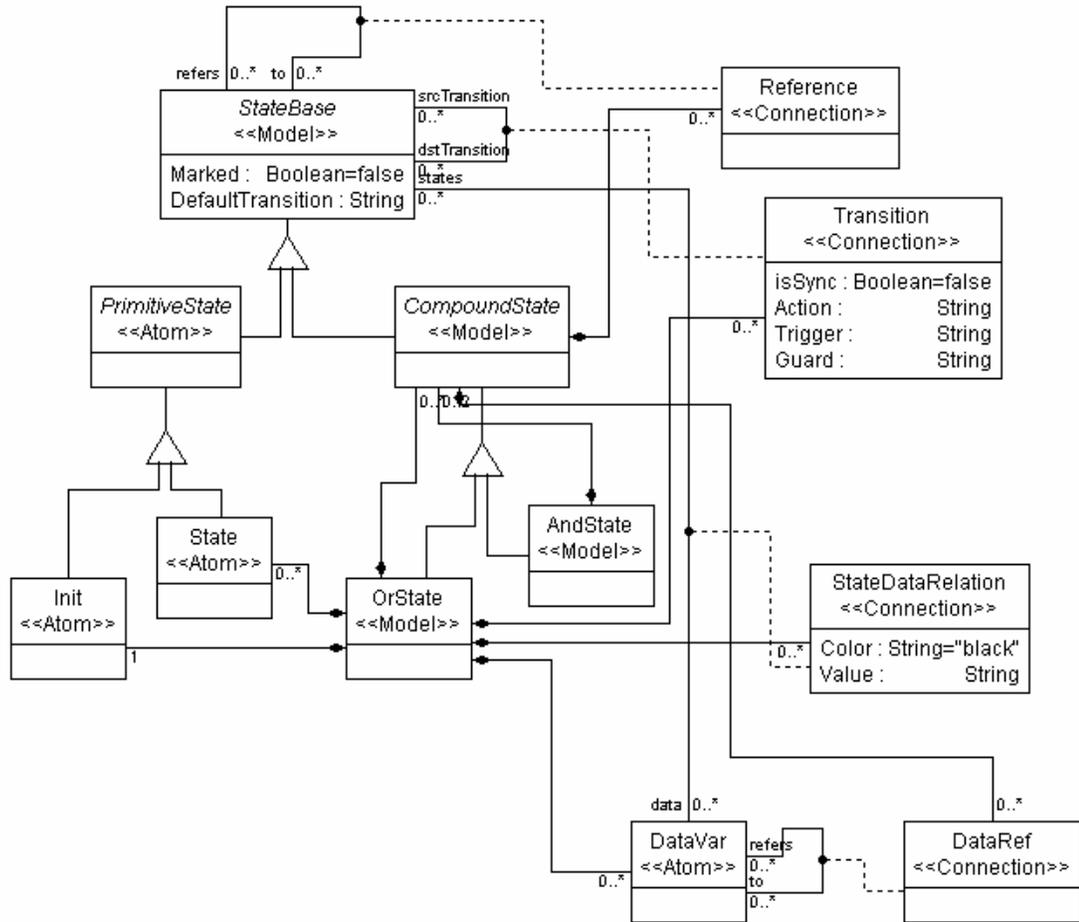


Figure 12: UML Meta model for the State Chart paradigm

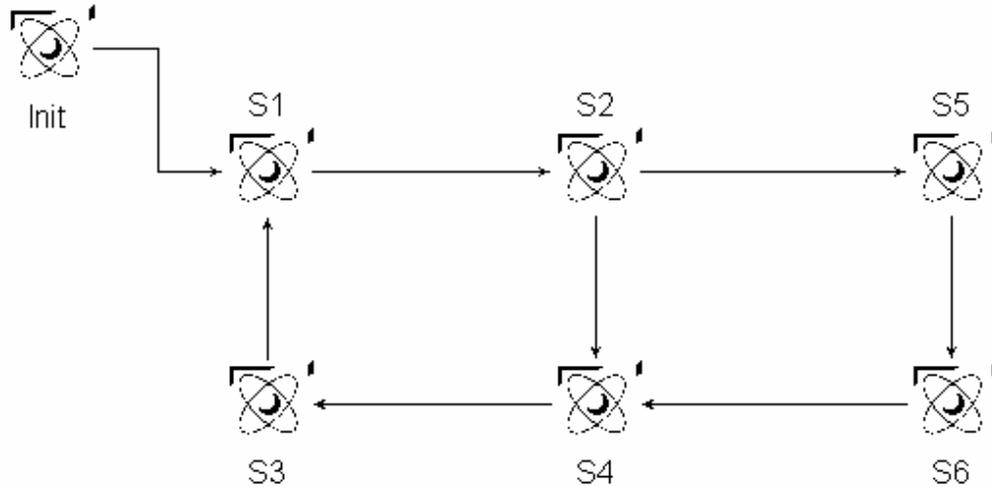


Figure 13: Example model for the State Chart paradigm

This is done by calling a function which will be implemented in another file, called *state.h*. The *state.c* file will be a constant for all models. The *state.h* file must be generated for each state machine model, to return a final state given the current state and the trigger value, which will be consistent with that model.

For simplicity, we will assume that there is only one “OrState” object containing a set of states and transitions, with no more levels of hierarchy. We will also assume that the names of the states and triggers are legal C++ identifiers.

Before any output is generated through a pattern file, we must open a file for output, and direct output to that file. The following lines achieve this, and will be the first lines of our pattern file:

```
<: open("state.h", o, statefile);
    switch(statefile); :>
```

These lines will open a file called “state.h” for output (overwriting any existing file with the same name), and directs output to that file. Note that the file name can be

any OCL expression that results in a string. Later examples will use this feature to create files with names taken from the model.

Once this has been done, we are ready to generate the actual output that will go into the file. Let us begin with some plain text that goes verbatim to the file.

```
/*
 * Finite state machine implementation
 * Generated by Pattern Processor for UDM
 */
#include <stdio.h>
#include <conio.h>
```

To keep things simple, this example will use numbers to identify triggers. The first step is to make a list of states and trigger values from the model and assign an integer value to each, using a series of #define statements.

The required output can be generated using the following statement:

```
#define <: print(s.name); :> <: print(num); :>\n
```

where “s” is a state instance and “num” is a number. This must be repeated for all the contained states. The following command will take a collection of “state” instances contained in an “OrState”, and print the above statement for each:

```
<: self.orState->forAll( o | {
    o.state->iterate(s; num : ocl::Integer = 1 | {
        :>#define <: print(s.name); :> <:print(num); :>\n<: num+1;
    } );
});
:>
```

This statement declares an iterator, which iterates over all “state” instances contained in an OrState “o”. The OrState is contained in the root object, which is represented by “self”. The iteration variable “num” is initialized to 1, and incremented by 1 for each iteration. This is done by the num+1 expression in the iterator, which must

always appear as the last expression wherever it is used. For the model shown in Figure 13, the output generated will be as below:

```
#define S1 1
#define S2 2
#define S3 3
#define S4 4
#define S5 5
#define S6 6
```

The following statement repeats the same thing for the triggers, with a small change:

```
<: self.orState->forAll( o | {
  o.transition->collect( t | t.Trigger; ).asSet()->iterate(t; num :
  ocl::Integer = 1 | {
    :>#define <: print(t); :> <: print(num); :>\n<: num+1;
  } );
} );
:>
```

It traverses all the transitions, and lists the triggers. In this case though, the trigger values can get repeated. The names of the states are names of the instances of the class State, and must be unique. The trigger is an attribute of the association class called Transition, and can have the same value across different instances of Transition. For instance, two transitions can have the same trigger value. Since we do not want these to get repeated, we first “collect” all the trigger values, and use the “asSet()” operation on it. This removes any duplicate elements. We then print a #define statement for each trigger. This defines integers 1, 2 etc. for the triggers.

The next step is generating the actual function which will return an integer representing the new state, given integers representing the current state and the trigger. The function heading will be plain text, as below:

```
int getNextState( int currState, int trigger) {
```

The function body must be of the form shown below (for the model in Figure 13):

```

if( currState = S2 ) {
    if( trigger = T2) return S5;
    if(trigger = T3) return S4;
    return -1;
}

```

The final state is returned on a successful transition, or -1 is returned indicating an error. A similar block must be generated for each state in the model. To achieve this, we must iterate through all the “state” objects in the model, and iterate through all the transitions from the state for each state object. This calls for a nested iteration. The following pattern script achieves this:

```

<: self.orState->forall(o | {
    o.state->forall( s | { :>\
        if( curState == <: print(s.name); :>) {<:
            s.transition[srcTransition]->forall(tr | { :>
                if( trigger == <: print(tr.Trigger); :> ) return <:
                    print(tr.dstTransition.name); :>; <:
            } ); :>
        return -1;
    } ); :>
} ); :>

```

This code first takes the top level “orState” and iterates through all the “state” objects that it contains. For each instance, it generates an `if (currState == statename)` statement. It then iterates through all the transitions originating from this state, using the `s.transition[srcTransition]` collection, generating an `if (trigger == triggername)` statement for each transition. Plain text is mixed in the pattern to insert the semicolons and other statements as necessary. A “\” at the end of a line prevents the new-line character from being printed at that point.

Finally, the function can be ended by returning -1 when all the comparisons fail. This follows as plain text at the end of the pattern file:

```

return -1;
}

```

The first example showed the usage of nested iterations, iteration variables, and simple mixing of plain text and pattern scripts. The next example will show the generation of output to multiple files and more nested iterations.

State Chart HTML example

This example will use the same state chart paradigm as the last example. The model however is a little more complicated. The model will contain several “orState” objects, each of which will contain a state machine. Our aim is to generate an implementation for each of the state machines in the model. The implementation will be an interactive system written in HTML and Java-Script.

An example model for this discussion is shown in figure 3. We will generate a separate HTML file for each system in the model, and finally generate an “index” file which will link all the other files. The first statement we must look at is opening the appropriate files for output. Unlike the previous example, the file names this time come from the model itself. The following script statements perform this task:

```
<: self.orState->forAll( oo | {  
    oo.compoundState->forAll( o | {  
        open(o.name + ".html", "o", stateFile);  
        switch(stateFile);  
    }  
}>
```

This script opens the top level “orState”, and iterates through the contained “orStates” (one for each contained state machine). It then opens a file for output, with the name of the state machine. The file name is derived from the OCL expression `o.name + ".html"`. A new file is opened in each iteration, and the output for that iteration is directed to that file.

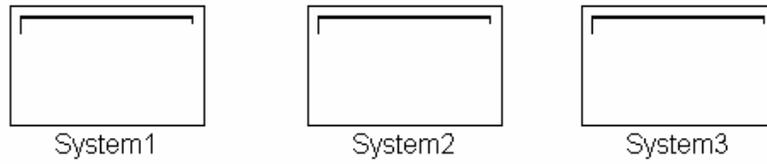


Figure 14: State Chart Model for example 2

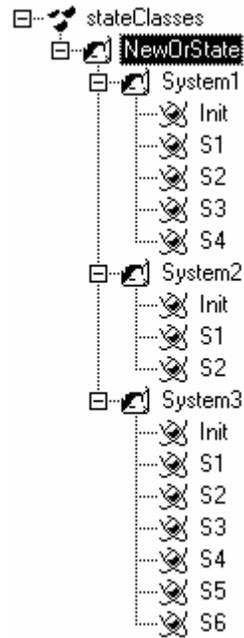


Figure 15: State Chart model for example 2 (Tree view)

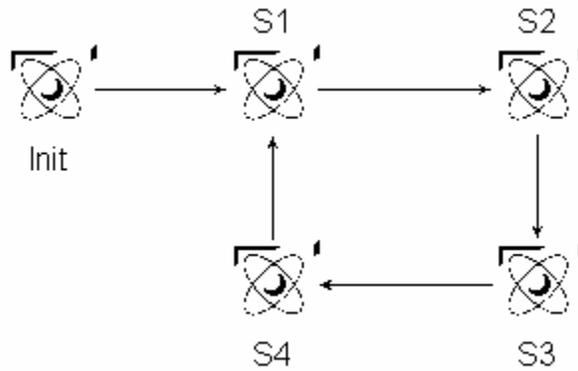


Figure 16: State machine for System 1 in Figure 14

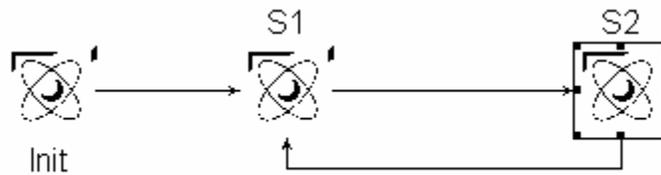


Figure 17: State machine for System 2 in Figure 14

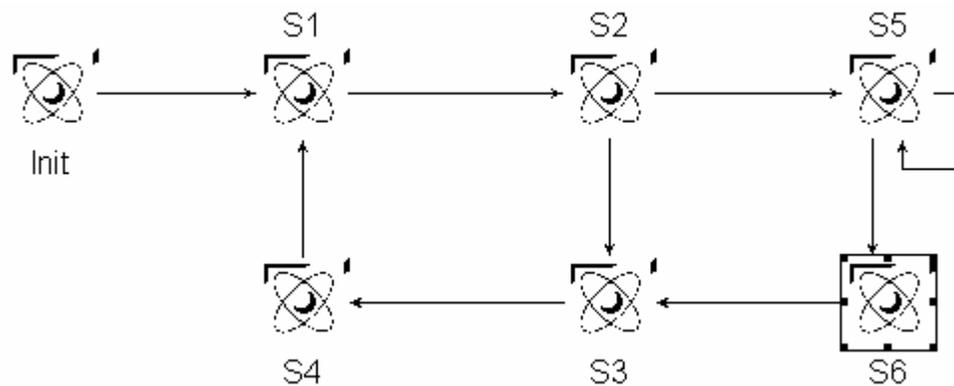


Figure 18: State machine for System 3 in Figure 14

In this example we generate a separate html file for each “orState” model. The html files will contain buttons for the available triggers, and clicking on a button will call a java-script function which will change the state. Buttons for triggers that are not available from a certain state will be disabled for that state. This is done by first calling a

function that disables all the buttons, and then the “setCurState” function which enables the relevant buttons. The code for the setCurState function is shown below:

```
function setCurState(state)
{
    disableAll();
    curState = state;
    dl.innerText = state;
<o.ooclAsType(OrState).state->forall( s | {
    :>\tif( curState == "<: print(s.name); :>" ) {\n<:
    s.transition[srcTransition]->forall(tr | {
    :>\t\tfrmTriggers.<:print(tr.Trigger);:>.disabled = false;\n<:
    });
    :>\t}\n<:
}); :>\
}
```

The script iterates through all the states contained in the orState “o”. This also demonstrates the usage of the oclAsType method. The top level iteration finds all “compoundState” objects contained in the top level “orState”, following the containment relationship shown in the UML diagram of the meta model in Figure 12. The generated JavaScript function first calls the disableAll() function which disables all buttons. Then it sets the current state using the innerText property of the DIV element in the HTML page [13]. The remaining portion of the generated code will enable the buttons for the triggers available from this state.

The rest of the HTML is generated in a similar way (the full pattern file can be found in the appendix). Once the output has been generated for all the state machines in the model, a final “index.html” file must be generated which will link to all the previously generated files. This is done by opening a new file as shown below:

```
<: open("statedemo.html", "o", main); :>
<: switch(main); :>
```

This is followed by the code which will generate the links, as shown below:

```
self.orState->forAll( oo | {
    oo.compoundState->forAll( o | { :>
        <a href="<: print(o.name); :>.html"><: print(o.name);
:;></a><br>\n<:
    });
});
```

This iterates through all the “orState” objects and creates an “<a href>” link for each iteration. The file name is generated similar to the way seen before for opening the files for output, using <: print(o.name); :> “html”. A screen shot of the HTML for System 3 in **Error! Reference source not found.** is shown below in Figure 19.

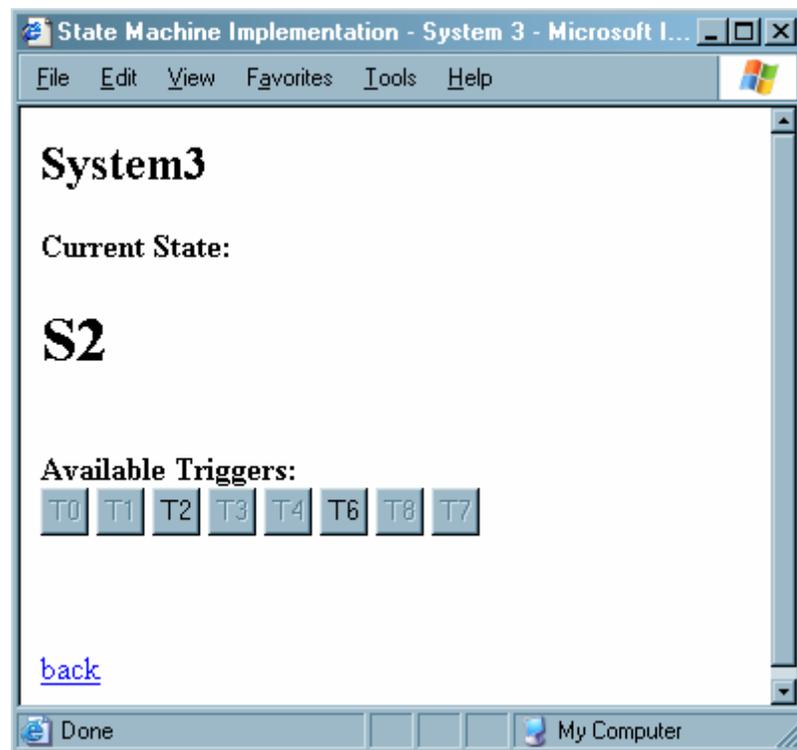


Figure 19: Sample output for example 2

Evaluation

In this section, we will compare the generation of a text file from a data network using other provided APIs, and see how it compares with using the pattern processor. For this study, we will rework the first example in this chapter.

The Generic Modeling Environment provides ways to write “interpreters” for the data networks created using GME. Interpreters are components that are loaded and executed by GME upon a user’s request. GME offers a network of C++ objects called the Builder Object Network (BON), which provide read/write access to the objects’ properties, attributes and associations [9]. The GME User’s Manual provides the details of using the BON.

Executing an interpreter from the GME GUI calls the “InvokeEx” method of that component. The implementation for this method in order to create an interpreter for the StateChart paradigm in Figure 12, which will generate an output similar to the one generated by the pattern file in Example 1 in this chapter, can be found in the Appendix.

This can be compared to the pattern file for Example 1 (the complete pattern file can be found in the Appendix). A quick glance shows that the interpreter code is about 58 lines long, while the pattern file is about 33 lines. The pattern file is almost half the size of the interpreter code, not counting the other files necessary to compile the interpreter file (which usually requires a set of header and .cpp files to create the component). This shows that even for a simple application, the pattern file provides a much more concise way of representing the required output.

In addition to the greater size, the interpreter code uses several classes defined in the BON such as CBuilderAtom, and several methods such as GetOutConnection etc.

These require the user to study the BON API and its usage, before attempting to construct the interpreter. These are difficult to learn for a user with little C++ experience. Moreover, these APIs are very specific to GME, and the information learnt here cannot be applied elsewhere. OCL on the other hand, is very easy to learn, and there are several books commonly available for learning OCL at various levels. Learning OCL will also be useful to any user involved in modeling, as OCL is becoming an inherent part of UML.

Finally, the interpreter must first be compiled and registered before it can be executed from the GME interface. When small changes are desired in the output, the C++ code must be changed, which requires the interpreter to be compiled and registered again. This becomes cumbersome when small corrections are to be made in the desired output.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

Conclusions

Model based programming is becoming increasingly popular, with new modeling tools and support applications being developed constantly. Modeling paradigms are developed very specifically for people who have specific domain knowledge but little programming knowledge. One of the main tasks in model-based applications is the generation of a text output from a network of models. There is a need for an easy-to-learn and easy-to-use application designed for this purpose.

The pattern processor developed in this study provides a simple interface to users for generating a text output from any UDM data network. It is generic and can be used with any kind of data network, based on any kind of meta-model. It uses a simple method of separating plain text and special script instructions, giving users a lot of control over spacing and layout in the desired output. OCL was chosen as the scripting language. Being tightly coupled with UML, OCL is very powerful for an application such as this. It is also very easy to learn, and has a high retention for users involved in modeling. Moreover, the navigation is done by directly using the class names in the domain, which provides an intuitive interface to the user. The examples show that it can be used to produce a wide range of outputs, such as C/C++ code, HTML files or XML files.

Using a pattern processor and a simple pattern file offers several advantages over hand coding an interpreter in C++ or any other programming language. While a C++

API for a modeling application can be very powerful, the simple OCL interface in the pattern processor offers all the constructs necessary to traverse a data network, retrieve data, and generate textual output. It is also a better alternative for a domain specialist who may not have an in depth knowledge of C++.

The pattern language described here separates the printing of plain text and data. This makes formatting the output an easier task, as the user can literally format it in a text editor, instead of adding escape sequences within C++ statements. It also makes the maintenance of the pattern files an easy task, as compared to maintaining a C++ project with several files, several thousand lines of code, and a large number of *printf* statements used purely to output constant text. The pattern script is interpreted when the pattern processor is executed. This takes away the need to compile the file for every small change. When generating textual output, small changes are made often to tweak small features in the output. Not having to compile the file for such small changes speeds up the development and eases maintenance. It has also been demonstrated that the output can be directed to multiple files easily, as can be done using C++.

Thus the new pattern processor developed in this project offers several advantages in accomplishing the task of generating textual output from models.

Future Work

Though the pattern processor is fully functional and can be used for most applications requiring generation of text output, there are several avenues for improvement. In the short run, the OCL parser needs to be updated to support some important OCL methods. For instance, the OCL parser used with this application does

not support the `sortedBy()` method. The power of the pattern processor is largely decided by the power of the OCL parser it uses. Improvements in the OCL parser will definitely improve the pattern processor. Improvements may be in the implementation of special functions which can enable a user to accomplish several specific tasks, or an improvement in the performance of the parser.

The next step is to couple this pattern processor with other modeling tools. It currently runs as a command line application and can interpret models in any of the back-ends supported by UDM. Integration with a popular modeling application such as GME will make its usage easier.

The pattern processor takes a data network and a pattern file, and produces a text output file, which may contain data retrieved from the data network. Another avenue for research is achieving the inverse. This involves taking a text file, which may contain arbitrary data, and a pattern file containing some rules based on which a data network can be generated, by retrieving data from the text file.

APPENDIX A

Pattern File for Example 1 (C++ Code Generation from State Chart model)

```
<: open("state.h", o, statefile);
    switch(statefile); :>
/*
 * Finite state machine implementation
 * Generated by Pattern Processor for UDM
 */
#include <stdio.h>
#include <conio.h>
<:
self.orState->forall( o | {
    o.state->iterate(s; num : ocl::Integer = 1 | {
        :>#define <: print(s.name); :> <: print(num); :>\n<: num+1;
    });

    o.transition->collect( t | t.Trigger; ).asSet()->iterate(t; num :
ocl::Integer = 1 | {
        :>#define <: print(t); :> <: print(num); :>\n<: num+1;
    });
});
:>\

int getNextState(int curState, int event)
{
<:
self.orState->forall(o | { :>\n<:
    o.state->forall( s | { :>\
        if( curState == <: print(s.name); :>) {<:
            s.transition[srcTransition]->forall(tr | { :>
                if( event == <: print(tr.Trigger); :> ) return <:
print(tr.dstTransition.name); :>; <:
            }); :>
            return -1;
        }
    });
};
:>
    return -1;
}
```

Sample Output generated by the Pattern File Above

```
/*
 * Finite state machine implementation
 * Generated by Pattern Processor for UDM
 */
#include <stdio.h>
#include <conio.h>

#define S1 1
#define S2 2
#define S3 3
#define S4 4
#define S5 5
#define S6 6

#define T1 1
#define T2 2
#define T3 3
#define T5 4
#define T6 5
#define T7 6

int getNextState(int curState, int event)
{
    if( curState == S6) {
        if( event == T3) return S6;
        return -1;
    }
    if( curState == S5) {
        if( event == T7) return S5;
        return -1;
    }
    if( curState == S4) {
        if( event == T2) return S4;
        return -1;
    }
    if( curState == S3) {
        if( event == T5) return S3;
        return -1;
    }
    if( curState == S2) {
        if( event == T6) return S2;
        if( event == T3) return S2;
        return -1;
    }
    if( curState == S1) {
        if( event == T2) return S1;
        return -1;
    }
    return -1;
}
```

Code for InvokeEx Method for GME Interpreter Equivalent to Pattern File in Example 1

```
void CComponent::InvokeEx(CBuilder &builder, CBuilderObject *focus,
CBuilderObjectList &selected, long param)
{
    const CBuilderFolderList *folds = builder.GetFolders();
    POSITION foldPos = folds->GetHeadPosition();
    while(foldPos) {
        ofstream outfile("state.h");
        outfile << "/*\n";
        outfile << " * Finite state machine implementation\n";
        outfile << " * Generated by Pattern Processor for UDM \n*/\n ";
        outfile << "#include <stdio.h>\n";
        outfile << "#include <conio.h>\n";
        CBuilderFolder *fold = folds->GetNext(foldPos);
        const CBuilderModelList *roots = fold->GetRootModels();
        POSITION rootPos = roots->GetHeadPosition();
        while(rootPos) {
            CBuilderModel *root = roots->GetNext(rootPos);
            const CBuilderAtomList *atoms = root->GetAtoms("State");
            POSITION pos = atoms->GetHeadPosition();
            int i = 100;
            while(pos) {
                CBuilderAtom *atom = BUILDER_CAST(CBuilderAtom,
                    atoms->GetNext(pos));
                outfile << "#define " << atom->GetName() << " " << i++ << "\n";
            }
            const CBuilderConnectionList *transitions =
                root->GetConnections("Transition");
            pos = transitions->GetHeadPosition();
            i = 200;
            CString tr;
            while(pos) {
                CBuilderConnection *transition =
                    BUILDER_CAST(CBuilderConnection, transitions->GetNext(pos));
                transition->GetAttribute("Trigger", tr);
                outfile << "#define " << tr << " " << i++ << "\n";
            }
            outfile << "int getNextState(int curState, int event)\n";
            outfile << "{\n";
            pos = atoms->GetHeadPosition();
            while(pos) {
                CBuilderAtom *atom = BUILDER_CAST(CBuilderAtom,
                    atoms->GetNext(pos));
                outfile << "\tif( curState == " << atom->GetName() << ") {\n";
                const CBuilderConnectionList *transitions =
                    atom->GetOutConnections("Transition");
                POSITION pos2 = transitions->GetHeadPosition();
                while(pos2) {
                    CBuilderConnection *transition =
                        BUILDER_CAST(CBuilderConnection, transitions->GetNext(pos2));
                    transition->GetAttribute("Trigger", tr);
                    outfile << "\t\tif( event == " << tr << ") return "
                        << atom->GetName() << ";\n";
                }
                outfile << "\t\treturn -1;\n";
            }
            outfile << "\t}\n";
        }
        outfile << "\treturn -1;\n";
        outfile << "};\n";
    }
}
AfxMessageBox("File state.h has been created", MB_OK | MB_ICONSTOP);
}
```

Pattern File for Example 2 (HTML Code Generation from State Chart model)

```
<:
self.orState->forall( oo | {
    oo.compoundState->forall( o | {
        open(o.name + ".html", "o", stateFile);
        switch(stateFile);
    }
}
:>
<!--
*****
Interactive HTML system generated from state chart model
using UdmOclPat

Author :    Ananth
Date   :    7 Nov, 2003
*****
-->

<html>
<title>State Machine Implementation - <: print(o.name); :></title>
<script language="javascript">
var curState = "";

function disableAll()
{
<:
    o.oclAsType(OrState).transition->collect( t | { t.Trigger; }
).asSet()->forall(t | {
    :>\tfrmTriggers.<: print(t); :>.disabled = true;\n<:
    });
:>\
}

function setCurState(state)
{
    disableAll();
    curState = state;
    dl.innerText = state;
<:
    o.oclAsType(OrState).state->forall( s | {
    :>\tif( curState == "<: print(s.name); :>" ) {\n<:
        s.transition[srcTransition]->forall(tr | {
            :>\t\tfrmTriggers.<: print(tr.Trigger); :>.disabled =
false;\n<:
            });
        :>\t}\n<:
    });
:>\
}

function activateTrigger(trigger)
{
<:
    o.oclAsType(OrState).state->forall( s | {
    :>\tif( curState == "<: print(s.name); :>" ) {\n<:
        s.transition[srcTransition]->forall(tr | {
            :>\t\tif(trigger == "<: print(tr.Trigger); :>")
setCurState("<: print(tr.dstTransition.name); :>");\n<:
            });
        :>\t}\n<:
    });
:>\
}
</script>
```

Pattern File for Example 2 (continued)

```
<body onload="javascript: setCurState('S1');">
<h2><: print(o.name); :></h2>
<b>Current State:
<h1><div id="d1"></div></h1>
<br>
<form name="frmTriggers">
Available Triggers:<br>
<:
    o.oclAsType(OrState).transition->collect( t | { t.Trigger; }
).asSet()->forall(t | {
    :><input type="button" name="<: print(t); :>" value="<:
print(t); :>" disabled=true onclick="activateTrigger('<: print(t);
:>')">\n<:
    });
:>\
</form>
</b><br>
<br>
<a href="statedemo.html">back</a>
<body>
</html>

<:
});
});
:>

<: open("statedemo.html", "o", main); :>
<: switch(main); :>
<!--
*****
Interactive HTML system generated from state chart model
using UdmOclPat

Author :    Ananth
Date   :    7 Nov, 2003
*****
-->

<html>
<title>State Machine Implementation</title>
<body>
<h3>Select OR state: </h3>

<:
self.orState->forall( oo | {
    oo.compoundState->forall( o | {
        :><a
href="<:print(o.name); :>.html"><:print(o.name); :></a><br>\n<:
        });
    });
:>
</body>
</html>
```

REFERENCES

- [1] The Object Constraint Language Specification, the Object Management Group.
- [2] Jos Warmer, Anneke kleppe, "The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition", Addison-Wesley Pub Co; 2nd edition, August 29, 2003.
- [3] OMG Unified Modeling Language Specification, Version 1.5, March 2003.
- [4] The Object Management Group, <http://www.omg.org>
- [5] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.
- [6] Arpad Bakay, Endre Magyari, "The UDM Framework", Institute for Software-Integrated Systems, Vanderbilt University, April 2003.
- [7] J. Craig Cleaveland, "Program Generators with UML and Java", Prentice Hall.
- [8] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley Pub Co, January 1986.
- [9] GME 3 User's Manual, Institute for Software-Integrated Systems, Vanderbilt University, March 2003.
- [10] Terrance John Parr, "Language Translation Using PCCTS and C++: A Reference Guide", Automata Pub Co, 1993.
- [11] Terrance John Parr, "ANTLR Reference Manual", January 20003, <http://www.antlr.org/doc/index.html>
- [12] J. Sztipanovits, and G. Karsai, "Model-Integrated Computing", IEEE Computer, Apr. 1997, pp. 110-112.
- [13] MSDN HTML and DHTML Reference http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/reference/dhtml_reference_entry.asp
- [14] Agrawal A., Karsai G., Shi F.: A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations, ISIS-03-403, November, 2003.
- [15] A. Nico Habermann, David Notkin, Gandalf: Software Development Environments, IEEE Transactions on Software Engineering, Vol. SE-12, No. 12, December 1986

- [16] David Notkin, The GANDALF Project, The Journal of Systems and Software, 5, 2
Special Issue on the Gandalf Project, 1985
- [17] Gail E. Kaiser, Elaine Kant, Incremental Parsing without a Parser, The Journal of
Systems and Software, 5, 2, Special Issue on the Gandalf Project, 1985
- [18] Frank Neven, Thomas Schwentick, Expressive and Efficient Pattern Languages for
Tree-Structured Data, International Conference on Management of Data and
Symposium on Principles of Database Systems, 2000
- [19] Yannis Smaragdais, Don Batory, DiSTiL: a Transformation Library for Data
Structures, USENIX Conference on Domain-Specific Languages, October 1997