

INTERPRETIVE PARSING TECHNIQUE
FOR BUILDING OBJECT NETWORKS

By

Nora Somogyi

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, 2005

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Sandeep Neema

ACKNOWLEDGEMENT

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 like to especially thank Dr Gabor Karsai and Dr Sandeep Neema for providing valuable feedback.

I would like to thank Feng Shi, Endre Magyari, Zsolt Kalmar and all the others in the MoBIES project for helping me whenever I encountered a problem.

TABLE OF COONTENTS

	Page
ACKNOWLEDGEMENT	ii
LIST OF TABLES	v
LIST OF FIGURES	vi
LIST OF ABBREVIATIONS.....	vii
Chapter	
I. INTRODUCTION	1
Existing technologies.....	2
Problems with the existing technologies	2
Requirements of a new technology.....	3
Problem Statement.....	5
Layout of Chapters.....	6
II. BACKGROUND.....	8
The Unified Modeling Language.....	8
Universal Data Model.....	9
Grammars and Language.....	12
Parsing and Parser Generation.....	16
III. MODELING CONTEXT-FREE GRAMMARS	18
Introduction.....	18
Designing the Metamodel.....	18
Decorator	19
Constraints	21
Implementing Regular Expressions	22
UDM object construction.....	23
Object creation.....	24
Attribute setting	25
Associations	26
IV. INTERPRETIVE PARSING	28
Introduction.....	28
Internal Operation of the Interpretive Parser Tool.....	30
Logging.....	30

Lexical Analysis.....	30
Grammar Analysis	31
Text file Analysis.....	32
UDM Object Network.....	32
Interpretive Parsing.....	32
V. EXAMPLES AND EVALUATION	34
State Chart.....	34
Time Series	38
Evaluation	43
VI. CONCLUSIONS AND FUTURE WORK.....	45
Conclusions.....	45
Future Work.....	45
Appendix	
A. THE PSEUDO-CODE DESCRIPTION OF THE PARSER ALGORITHM	47
B. GRAMMAR DESCRIPTION OF STATE CHART TEXT	48
C. THE STATE CHART TEXT.....	50
D. THE OUTPUT STATE CHART MODEL.....	51
E. THE OUTPUT UDM STATE CHART OBJECT NETWORK.....	52
F. GRAMMAR DESCRIPTION OF SIMPLE TIME SERIES.....	53
G. GRAMMAR DESCRIPTION OF TIME SERIES WITH DATA TRIPLETS.....	54
H. THE SIMPLE TIME SERIES TEXT	55
I. THE OUTPUT UDM SIMPLE TIME SERIES OBJECT NETWORK	55
J. THE TIME SERIES TEXT WITH TRIPLETS	56
K. THE OUTPUT UDM TIME SERIES WITH TRIPLETS OBJECT NETWORK	56
REFERENCES	57

LIST OF TABLES

Table	Page
1. An example context-free grammar	13
2. Rewrite the start symbol “program”	13
3. The EBNF version of the example grammar	15

LIST OF FIGURES

Figure	Page
1. The workflow of generating a UDM network from text.....	3
2. The inputs provided to the interpretive parser tool	6
3. The basic elements of UML class diagrams.....	9
4. The UDM Framework.....	10
5. C++ classes generated by UDM [4]	11
6. The parse tree of the derivation.....	14
7. The syntax diagram of the example grammar	15
8. Direct mapping to/from EBNF	16
9. The context-free grammar metamodel	20
10. An example rule	22
11. Object hierarchy in a class diagram	25
12. Simple association in a class diagram	26
13. Association class in a class diagram	27
14. Association via association class.....	27
15. The six steps performed by the parser.....	30
16. GME metamodel for the State Chart paradigm.....	34
17. Example model for the State Chart paradigm	35
18. The UML metamodel of the State Chart paradigm.....	36
19. UML metamodel for Time Series	39

LIST OF ABBREVIATIONS

API	Application Programming Interface
ANTLR	ANother Tool for Language Recognition
BNF	Backus-Naur Form
DTD	Document Type Definition
EBNF	Extended Backus-Naur Form
BON	Builder Object Network
DTD	Document Type Definition
GME	Generic Modeling Environment
IDE	Integrated Development Environment
ISIS	Institute for Software Integrated Systems
MIC	Model Integrated Computing
MIPS	Model Integrated Program Synthesis
OCL	Object Constraint Language
UDM	Universal Data Model
UML	Unified Modeling Language
XML	eXtensible Markup Language
XSD	XML Schema Definition
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformation

CHAPTER I

I. INTRODUCTION

Numerous modern computer-based systems are large, complex, heterogeneous, and mission-critical. The modifications of these systems involve risk that is proportional to the size of the system and not to the size of the change; for that reason, systems must be designed to evolve. There are many key factors that must be considered when designing these systems such as application evolution and design environment evolution, and MIC (Model Integrated Computing) can be used to facilitate this evolution. MIC is a methodology used to create and evolve integrated, multiple aspect models of computer-based systems using concepts, relations, and model composition principles to facilitate system/software engineering analysis of the models and automatic synthesis of application from the models [1]. A model is an abstract representation of an object or a concept; moreover, models can represent not only individual entities but also a network of objects. However, there are many cases when only a textual representation of an object network exists, and converting the textual representation into object networks is a problem often encountered.

Many model-based tools are offered on the market, both academic and business, that can be used to build and manipulate large object network models. Unfortunately, these tools usually do not provide reusable mechanisms that can convert text to data structures; however, they usually do provide processes to extract and format text from the object networks. Some tools for creating and manipulating such network models are developed at the ISIS (Institute for Software Integrated Systems), such as GME (Generic Modeling Environment) [3], UDM (Universal Data Model) [4], and GReaT (Graph Re-writing and Transformation Engine) [5].

There is a need for converting text into data structures in a convenient way. In this thesis, interpretive parsing techniques are introduced that provide a convenient way to parse a text file and interpret the corresponding object construction actions defined in a grammar to generate data structures from the text file.

Existing technologies

The UDM (Universal Data Model) framework includes the development process and a set of supporting tools that are used to generate C++ programmatic interfaces from UML class diagrams of data structures. UDM can be best used where object-oriented approach is followed to describe the data structures [4]. UDM object networks are defined by class diagrams, and the objects in the network are instances of classes. Similarly to UML, objects can have attributes, and associations among objects can be defined. UDM includes several generic programs that work on UDM data, such as the declarative pattern processor called UdmOclPat that interprets UDM data network and generates text output through a simple OCL-based query language [6].

Although UdmOclPat provides a standard and convenient way to generate text output from a data network, there is no standard way to close the loop and generate data network from text. However, users can come up with different procedures to achieve this result. One common way is to define a grammar for the text file in a well-known format (i.e., in EBNF), use a parser generator tool such as ANTLR to generate a parser, and modify the generated parser's source code to execute UDM specific commands. Similarly, we can also write an XSLT script that transforms a given XML to a specific XML format that complies with UDM's XML data format, but the trick is basically the same: we define a grammar and a parser in the script.

Problems with the existing technologies

Writing grammars and generating and modifying parsers is cumbersome and error prone if the format and the content of the input text often change. The problem we have to face is that whenever the structure of the text changes, we have to write a new or modify the existing grammar, generate a new parser, and browse and modify the source code of the generated parser to reflect the changes in the structure of the text.

However, users can use MIC to alleviate this cumbersome work by creating a GME paradigm to build context-free grammars by drawing syntax diagrams. Staying with the example, users can add ANTLR and UDM specific attributes to their paradigm and use BON [3] to write an interpreter that generates an ANTLR specific grammar file from the model. After generating the parser, there is no need to modify the source code provided that the built-in UDM specific commands can be executed. However, the users cannot take the UML class diagram, i.e., the metamodel of the UDM network for granted, either the UDM meta should exist or the users

should create them in advance. The parser cannot build an object network from the text without the UDM meta information. As a result, the paradigm-dependent API files (.cpp, .h, and .xsd) also should be generated. The workflow is shown on Figure 1.

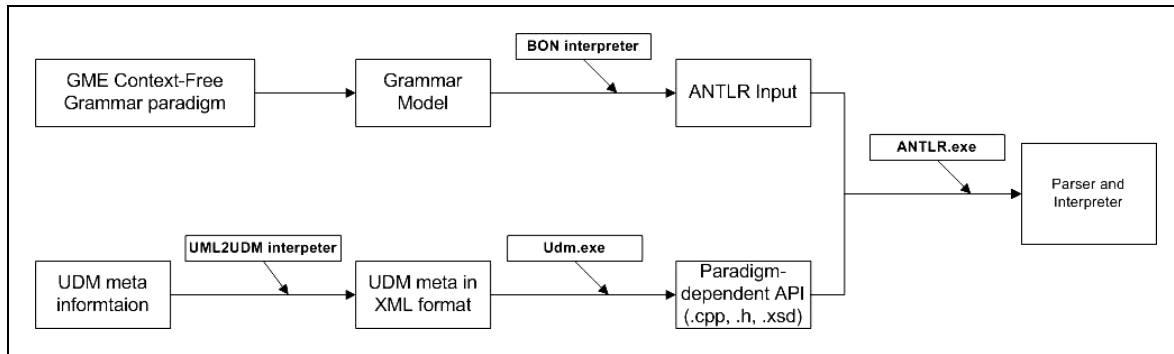


Figure 1 The workflow of generating a UDM network from text

This compilation-based approach requires a deep knowledge of not only ANTLR but also C++ STL and UDM. Moreover, this process is fragile because there are many code generation steps, and if one fails, the others could fail, too, and thus, extra effort is needed to find the failing step in the workflow. Another important problem with this approach is that everything must be regenerated, recompiled, and linked even for small changes.

Requirements of a new technology

Converting text to an object network is a problem often arises; thus, there is a need for a new technology that is easy to use and provides a convenient interface to extract data from the text and generate a UDM object network. A grammar editor tool and a simple interpretive parser that processes the text file and the constructed grammar is the right choice for solving the problem. Generally, the process of computing the structural description for an overt form is called interpretive parsing. In this case, interpretive parsing means to compute the structural description of a UDM network from a given grammar and an arbitrary text.

Grammar editors are not widely offered because they are not general in the sense that a grammar editor is either specific to a language or to a tool that uses the constructed grammar. For example, ANTLR [16] uses grammars written in the EBNF meta language to generate specific

parsers to specific grammars. For that reason, many plug-ins are offered for already existing popular IDEs, such as the ANTLR-aware multi-page text editor with syntax highlight [17] for Eclipse [18]; however, no visual builder tools are offered that facilitate such grammar construction. Although some visual builder tools exist, these tools are specific to or part of a toolset, such as the Visual Grammar Builder Studio [19] that is used in the speech technology.

The required features of a grammar editor tool must be the following:

- It must offer the construction of context-free grammars. Context-free grammars are simple enough to allow the construction of efficient parsing algorithms to determine whether and how a given text corresponds to the grammar.
- It must offer a simple but powerful visual editor to open, create, edit, and save grammars. A unique design view framework could display any context-free grammar, and GME provides a way to create this framework. Customizable visual icons are simpler to read and understand than a simple text itself. In addition, copy, paste, import, export, and syntactic and semantic checks are already available in GME that could further ease the development of these grammars.
- It must allow the user to conveniently assign UDM object construction actions to the structures of the grammar. An interpretive parser can interpret and execute these actions to build UDM networks.
- It must ease the design of the grammar by providing concise messages when detecting ambiguities and other errors in the grammar during construction.

The parser generator ANTLR compiles the object construction actions found in its grammar description into the generated parser, so the user has to modify the source code if necessary and has to recompile and re-execute the parser. In contrast to this approach, our interpretive parser interprets the UDM object creation actions included in its grammar description. Since there are no intermediate source files to compile, the user does not have to deal with compilation or linking.

The required features of an interpretive parser tool must be the following:

- It should work as a recursive descent LL(k) parser. Recursive descent parsers recognize the class of LL(k) grammars for unbounded k. Although recursive descent parsers lack speed, they have the advantages that they are simple yet powerful can pass information down to subrules, and subrules can pass

information up, allowing for parameterized non-terminals. These advantages will be used when interpreting UDM object construction actions of the grammar.

- It must offer early detection mechanisms of immediate or indirect left-recursion [14] in the grammar because a formal grammar that comprises left recursion cannot be parsed by a recursive descent parser. The running time of a recursive descent parser is exponential, and a left recursion would result in an unbounded recursive loop that would cause abnormal program termination.
- It must provide concise error messages when detecting lexical and syntactic errors in the text or syntactic and semantic errors in the grammar. The user must be notified where and why the text does not correspond to its grammar or the grammar construction is not correct.
- It must provide concise error messages when the interpretations of UDM object construction actions fail. The user must be notified where and why an interpretation fails.
- It must be a simply and easy to use library to avoid complexity and other intermediate compilation or linking processes.

Problem Statement

My objective is to develop a visual context-free grammar editor tool and an interpretive parsing tool.

The grammar editor tool facilitates the design of a grammar visually, and no other inputs are needed for this tool. The output of this tool is an MGA or XML file that serves as an input to the parsing tool.

The interpretive parsing tool takes an input text file, an input grammar file, and a destination UDM object network. After parsing the text file and interpreting the object creation actions from the grammar, the parser tool returns the modified UDM network as an output. As Figure 2 shows, the inputs provided to the tool will be as follows:

Text file: This text file can contain arbitrary plain or formatted text.

Context-free grammar file: The file contains a grammar model created in GME [3] using the context-free grammar paradigm. The tool supports two of the three persistence technologies of UDM, so this is usually in an MGA or XML file.

UDM network: This object network must be created with its root object in advance by the user. We assume that the user of this parser tool already has the UML class diagram, the UDM network, and the necessary framework to manipulate the UDM network.

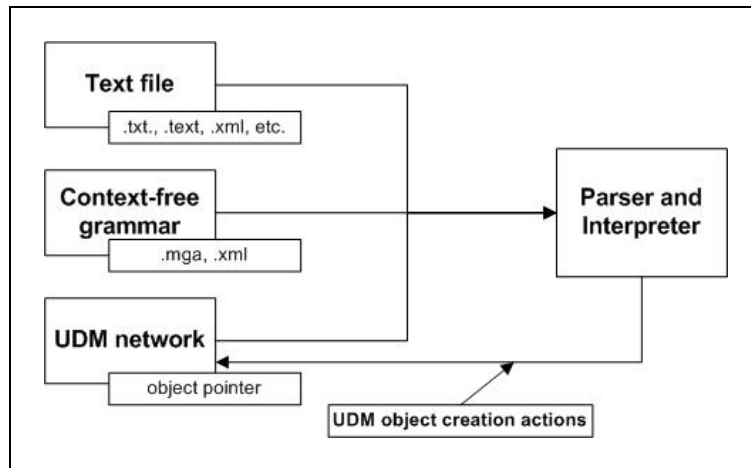


Figure 2 The inputs provided to the interpretive parser tool

The interpretive parser tool will return the modified UDM network after performing the following steps providing that no errors or exceptions occur:

- Read in the grammar file and check for ambiguity and left-recursion
- Configure the built-in recursive descent parser
- Read in the text file
- Open the UDM network and check for class diagram and root object
- Start the lexer, parse the text file, and interpret and execute the built-in UDM object construction actions

In case of I/O or other parsing errors, the tool will stop and give either a concise error message or a suggestion on how to correct the error or both.

Layout of Chapters

Chapter 2 contains an introduction to the standards and tools that are used in this thesis, such as UML, UDM, grammars, and parsing techniques. Chapter 3 and 4 follows with the architectures of the grammar editor and the interpretive parser tools and explains the motivation

behind the chosen techniques. Chapter 5 will give a real life example for using the interpretive parser and compares its effectiveness to a hand-written application that can be used only on one type of text input and UDM meta. Chapter 6 closes the thesis by drawing conclusions and showing how related future work can improve this application.

CHAPTER II

II. BACKGROUND

The Unified Modeling Language

The Unified Modeling Language (UML) is a language for specifying, visualizing, and documenting models of software systems, including their structure and design, in a way that meets all of these requirements. UML can be used for not only software modeling but also modeling of other non-software systems. UML is an OMG (Object Management Group [8]) standard, and the members of this not-for-profit computer industry specifications consortium define and maintain the UML specification. [7]

UML represents a unification of the concepts and notations of Booch, Rumbaugh, and Jacobson [10]. Notation plays an important role in modeling and the goal of the UML is to serve as a universal notation for creating models of object-oriented software.

Many UML-based tools exist on the market, such as Rational Rose [9], and they can be successfully used to analyze the requirements of the system and design a solution that meets them. UML 2.0 [7] offers twelve standard diagram types to represent these results.

UML includes the following components for building object-oriented and component-based systems [7]:

- Model elements — fundamental modeling concepts and semantics
- Notation — visual rendering of model elements
- Guidelines — idioms of usage within the trade

Class diagrams are the most commonly used artifacts of UML, and they represent the static structure of the classes and their relationships, such as inheritance and aggregation, in a system. UML class diagrams describe three different perspectives when designing a system, such as conceptual, specification, and implementation. Class diagrams model class structure and contents using design elements such as classes. Figure 3 [11] shows the basic elements of a UML class diagram.

A class is represented by a rectangular icon divided into three parts: class name, attributes, and operations. Attributes are listed along with their type and access modifiers, and operations are listed along with their return types, parameters, and access modifiers.

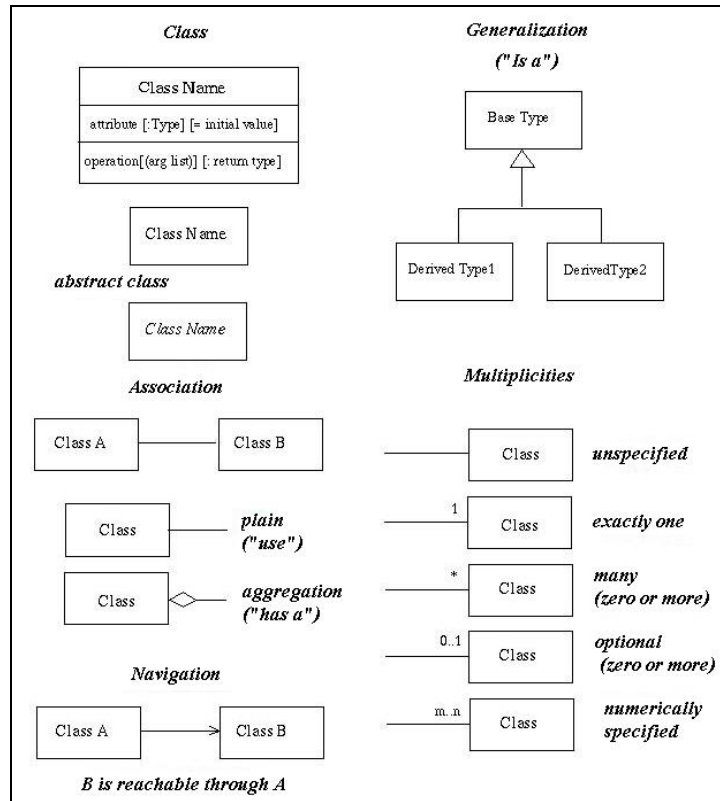


Figure 3: The basic elements of UML class diagrams

Associations represent relationships between instances of classes, and each association has two roles; each role is a direction on the association. We can indicate how many objects may participate in a given relationship by assigning multiplicity to the role. Multiplicity indicators are 1 (exactly one), * (zero or more), 1..* (1 ore more), 0..1 (0 or 1), and so on. Generalization implies inheritance, and this relationship is an association with a small triangle next to the class being inherited. In contrast, an aggregation implies a part/whole relationship, and this relationship is shown as an association with a diamond next to the class representing the aggregate.

Universal Data Model

The UDM (Universal Data Model) tool suite defines the development process and includes a 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 storage services for data structures as described in the input UML diagram [4].

Currently three persistent storage technologies are supported as follows:

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

UDM can be best used where the object-oriented approach is followed to describe the data structures. First, the object structure must be defined in the form of a class diagram, then the components of these class diagrams, such as classes, attributes, and associations can be accessed through the C++ API framework UDM generates. The generated API offers a convenient way for building and manipulating object networks, and it also provides interfaces for navigation. The UDM includes the following generic tools:

- UML paradigm for GME with a built-in interpreter
- UDM framework generator to generate the paradigm-dependent API files
- UdmCopy to port data between different persistent technologies
- UdmOclPat to interpret UDM data and generate text output

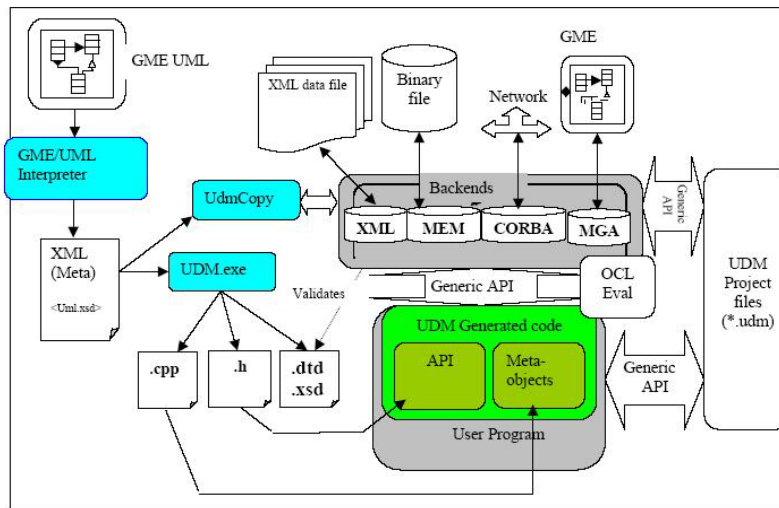


Figure 4: The UDM Framework

Figure 4 describes the UDM framework. The typical process of using UDM starts with a UML class diagram created in GME by using the provided UML paradigm. Since the internal format of GME is not compatible with UDM, we have to use the built-in interpreter to convert the UML diagram to UDM diagram. The output is an XML file. Next, we can generate the paradigm-dependent API files by passing the XML file from the previous step to the UDM generator. The output is a header (.h) and a program file (.cpp) along with DTD files (.dtd, .xsd) that correspond to the UML class diagram. The user can build a C++ project by including these generated files and the UDM specific libraries and headers to provide a convenient API to access components of a data structure based on the initial class diagram. However, the UDM provides the generic interface TOMI that can be used to manipulate UDM data without the domain specific, generated API, but this paradigm independent interface is not as efficient as the domain specific, generated one.

The UDM API maps each UML class in the source diagram to a C++ class with the corresponding name, and the mapping preserves namespaces. Figure 5 shows an example for how UDM maps simple classes to UDM classes. As we can see from the picture, the generated classes have a superclass in UDM, the `Udm::Object`. Every generated class derives from this class, which also defines generic access and navigation interfaces. Object creation follows the factory method design pattern; for that reason, static factory methods are provided for creating objects for these classes.

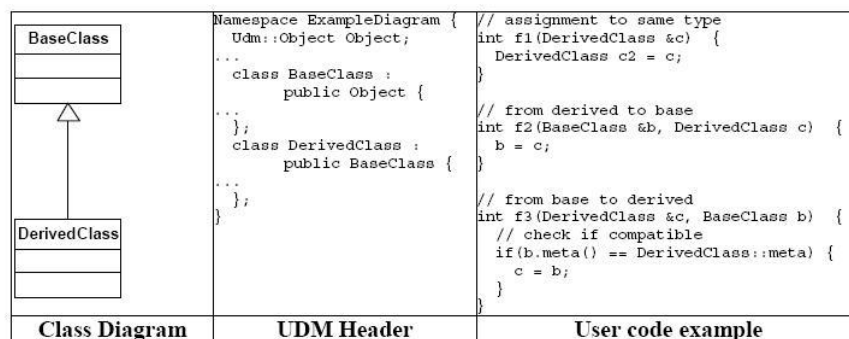


Figure 5: C++ classes generated by UDM [4]

The UML attributes are also mapped to C++ members along with access methods to them, but all attributes are considered to be public. The mapping preserves the inheritance relationships of the UML diagram by converting them to public inheritance in C++. UML containment relationships are translated into access methods for both the child and parent classes. Containment is the most important relationship in UDM, and all objects must be contained in exactly one parent, and this relationship can be changed by using the access methods provided by the generated API.

Grammars and Language

Any natural or programming language has a vocabulary of words that can be used to assemble legal sentences in the language. But not all possible sequences over a given vocabulary are legal, as the sequences need to meet certain syntactic and semantic restrictions. Grammars provide a method for specifying whether the sentences in a given language are syntactically legal or not.

Example:

- Legal C++ statement: `int i = 1;`
- Illegal C++ statement: `int for = i;`

A generative grammar is defined by sets of terminal and non-terminal symbols, a start symbol, and a set of production rules [15]. A grammar G is a structure of (N, T, P, S) where N is a set of non-terminals, T is a set of terminals, P is a set of productions, and S is a special non-terminal called the start symbol of the grammar. A formal language is a set of finite-length words, such as character strings, drawn from some finite alphabet. The containment hierarchy of classes of formal grammars that generate formal languages are defined by Chomsky [15]. The Chomsky hierarchy consists of four levels, such as the type-0 unrestricted grammars, the type-1 context-sensitive grammars, the type-2 context-free grammars, and the type-3 regular grammars.

In a context free grammar, each rule consists of a left-hand side and a right-hand side, where the left-hand side can be only a non-terminal symbol. The right-hand side of a rule consists of a (possibly empty) sequence of terminal and non-terminal symbols separated by spaces and operations that allow concatenation, alternation, repetition, and grouping of symbols. Each rule can be viewed as a rewriting rule, specifying that whenever we have an occurrence of a

left-hand symbol, it can be rewritten to the right-hand symbol. Table 1 shows an example context-free grammar that describes how to write a legal sentence in a simple language.

Table 1 An example context-free grammar

Left-hand side	Right-hand side
program	Block
block	(statement)*
statement	Assignment while_stmt
assignment	Identifier "=" expression
expression	Identifier number
while_stmt	"while" expression "do" block "end"
Identifier	(a-z A-Z)+
Number	(0-9)+

As we can see, there is a start symbol defined for this grammar, and we can rewrite this non-terminal symbol `program` as shown on Table 2:

Table 2 Rewrite the start symbol "program"

program	
	• block
	• statement
	• assignment
	• identifier "=" expression
	• identifier "=" number
	• "a123" "=" "456"

The above rewrite sequence is also called as derivation. This derivation maps sentence shown on Table 2 to a structure, and the derivation can be represented with a derivation or parse tree, see Figure 6.

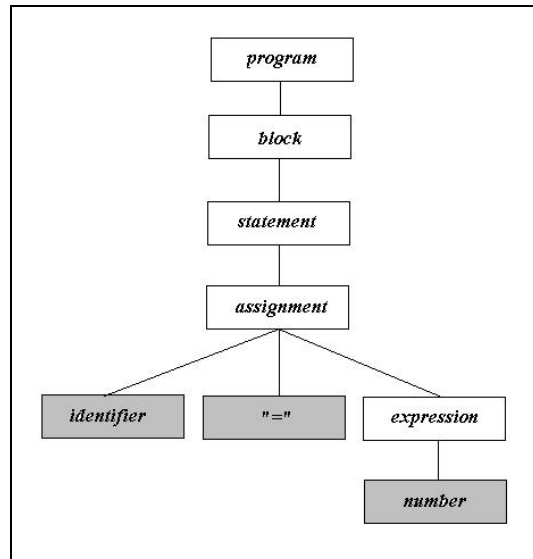


Figure 6 The parse tree of the derivation

The root of the parse tree corresponds to the start symbol `program`. In each step in the derivation, we replace a non-terminal with its corresponding right-hand side. The symbols of this right-hand side are added as children to the node. If the derivation is complete, we can read the result from the tree because the leaves correspond to the terminal symbols in the original sentence. The left-to-right order should be followed when reading the leaf nodes.

The Backus-Naur form (BNF) (also known as Backus normal form) is a metasyntax used to express context-free grammars, i.e., a formal way to describe formal languages. BNF is widely used as a notation for the grammars of computer programming languages, command sets, and communication protocols. For historical reasons there are many variants of BNF, and EBNF is one of them. The EBNF specification consists of terminal and non-terminal symbols that form derivation rules. Terminal symbols can be constant tokens, constant characters, and regular expressions over constant characters. A special of the non-terminal symbol is the start symbol that denotes the non-terminal symbol from which the derivation starts. Parentheses are also used to enhance readability. EBNF defines the following extensions to BNF:

- Kleene cross: $()^+$, a sequence of one or more elements
- Kleene star: $()^*$, a sequence of zero or more elements
- option: $()?$, the element is optional

The example provided on Table 1 can be rewritten in EBNF form as follows on Table 3.

Table 3 The EBNF version of the example grammar

```

program := block
block := (statement)*
statement := assignment | while_stmt
assignment := identifier EQUAL expression
expression := IDENTIFIER | NUMBER
while_stmt := WHILE expression DO block END
WHILE := "while"
DO := "do"
END := "end"
IDENTIFIER := (a-z|A-Z)+
NUMBER := (0-9)+
EQUAL := "="

```

Syntax diagrams are also used to express context-free grammars, and sometimes these are preferred because they are intuitively more readable than the corresponding EBNF grammar since they enable a better overview of the connectivity of concepts in the grammar. Syntax diagrams are like flow charts, and they can be read from the left side to the right side, following the arrows. Terminal symbols are represented with ellipses, and non-terminal symbols are represented with rectangles. Figure 7 shows the syntax diagram that corresponds to the example grammar above.

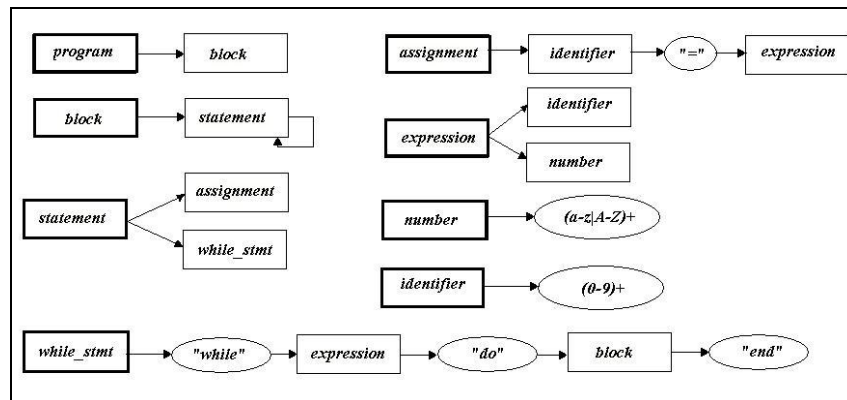


Figure 7 The syntax diagram of the example grammar

The direct mapping of syntax diagram to/from EBNF is summarized on Figure 8.

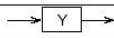
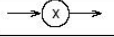
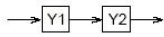
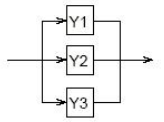
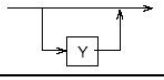
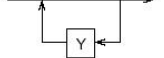
nonterminal	Y	
terminal	x	
sequence	Y1 Y2	
alternation	Y1 Y2 Y3	
optional	[Y]	
repetition	{ Y }	

Figure 8 Direct mapping to/from EBNF

Parsing and Parser Generation

The sentences of a language can be analyzed by extracting the grammatical structure of the sentence, which is done by building a parse tree for the sentence. The language analysis of a system consists of two consecutive steps, namely lexical and syntactic analysis.

A lexical analyzer, or in other words a scanner, is a program that takes a stream of characters as input and outputs substrings it recognizes. Each substring is referred to as a token and has a usually numeric identifier assigned to it to uniquely identify its type. [12] Assume that we have an input string "a0 ben 123", and we want our lexical analyzer to recognize "a0" (a letter followed by a digit), "ben" (a sequence of letters), and "123" (a sequence of digits), and the spaces also have to be explicitly recognized. We use regular expressions to define each token, so we can describe the tokens above as follows:

- [a-z][0-9] - a letter followed by a digit
- [a-z]+ - a sequence of letters
- [0-9]+ - a sequence of digits

The lexical analyzer returns the tokens "a0", "ben", and "123" in the knowledge of the definitions of each token. The scanner has to recognize an unknown token, and it also has to recognize the end of the input [12].

The syntactic analysis, or in other words the parsing, recognizes valid sentences of a language by analyzing the syntax structure of a set of tokens passed to it from a lexical analyzer. The parser accepts a stream of tokens from the scanner, extracts the grammatical structure of the sentence, and builds a parse tree for the sentence. The parse tree can be constructed in two different ways: top-down and bottom up [14]. In top-down or LL-parsing the parse tree is constructed from the top down; in other words, the parser attempts to build a derivation of the sentence from the start symbol of the grammar by replacing non-terminal symbols with their corresponding right-hand side. In bottom-up or LR-parsing the parse tree is constructed from the bottom up; in other words, the parser attempts to build a derivation from the sentence to the start symbol. The difference between these two parsing methods lies only in the construction. However, bottom-up method has more powerful recognition capability because it delays parsing decisions as long as possible, but translation or compiling is generally easier with top-down parsing. Both parsers make the same decisions when constructing their parse trees, namely which non-terminal to choose when it rewrites a rule and which left-hand side rule to use when it expands the chosen non-terminal. The parser usually needs to look at one or more tokens to make its parsing decisions, and these tokens which the parser is examining but not yet consumed are called lookahead tokens [13]. A top-down parser that needs to use a single lookahead token is called LL(1) parser, and if it needs k lookaheads, then it is an LL(k) parser. Similarly, a bottom-up parser that needs to use a single lookahead token is called LR(1) parser, and if it needs k lookaheads, then it is an LR(k) parser.

We can automate the process of parsing by generating a parser from the grammar. An advantage of using a parser generator is that it verifies that the grammar is suitable to the parsing method being used. In top-down parsing, this results in the form of a recursive-descent program. The recursive-descent parsing is based on recursive subroutines: a subroutine is written for every non-terminal in the grammar, and these subroutines call each other in the order specified by the grammar to simulate traversing a parse tree. The most popular parser generator tool is the ANTLR [16]. ANTLR, ANOther Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, Python, or C++ actions. ANTLR is easy to understand, powerful, flexible, and generates human-readable output; moreover, ANTLR provides excellent support for parse tree construction, tree walking, and translation [16].

CHAPTER III

III. MODELING CONTEXT-FREE GRAMMARS

Any text can be generated from a UDM network by using the UdmOclPat program that interprets UDM data network and generates text output through a simple OCL-based query language [6]. This thesis addresses the goal to close the loop, i.e., to generate a UDM network from a text file providing that the text file is grammatically correct. For that reason, a grammar editor tool is needed, and the grammar is used to check the syntax of the text. The grammar can be written by hand; however, this is tedious and cumbersome work because the grammar in a text format is not intuitive and does not provide a good overview of the connectivity of concepts in the grammar. This chapter focuses on this problem, i.e., it describes an environment to model grammars.

Introduction

A grammar can be represented not only textually but also visually by syntax diagrams, and GME provides a way to model the syntax diagram via a context-free grammar editor. Although GME provides a mechanism to extract semantics from a model, we do not use this feature because the graphical representation serves as an input to the interpretive parser tool developed in the second part of the thesis.

Designing the Metamodel

The metamodel models context-free grammars that define syntax rules in terms of terminal symbols (the elements of the source text) and non-terminal symbols (the syntactical categories of the source language) and supports alternative definitions and recursion. A grammar specification is a set of derivation rules are restricted: only non-terminal symbols are accepted on the left side of a grammar rule, and a special non-terminal called the start symbol must be also defined for a grammar.

The construction of the metamodel is straightforward from the descriptions of the syntax diagrams. Symbols are GME atoms and the GME arrows are connections in the metamodel. We need three types of symbols in the metamodel: terminal symbols, non-terminal symbols, and

references to non-terminal symbols. We distinguish three different types of terminal symbols, namely constant tokens, constant characters, and regular expressions. Both terminal and non-terminal symbols are GME atoms in the metamodel, but references to non-terminal symbols are GME references.

A rule in the grammar is described as graph formed from symbols and connected with directed arrows; thus, connections among symbols, among symbols and references, and among references are defined. The context-free grammar metamodel, called CFG_ML is shown on Figure 9. In the hierarchy of a grammar model terminal and non-terminal symbols and rules would be created in separate containers, and both terminal and non-terminal symbols must be predefined before building up rules.

Decorator

GME provides a mechanism, called decorator, to improve and customize the visual representation of the models. Although the presentation can be improved with icons, there are certain circumstances when icons cannot provide enough information; for example, the icon should change when certain property of a model element is modified. This problem can be solved by using decorators in GME. Decorator components are assigned to the classes of the metamodel; therefore, the default appearance of the objects in the model can be overwritten. In our metamodel, non-terminal symbols are represented with rectangles, and the name of these symbols is written inside the rectangle. Terminal symbols are represented with ellipses, and the value property of these symbols is written inside the ellipse and the name of these symbols is written under the ellipse. In case of a reference to a non-terminal, the name of the referred object is written inside a red rectangle, and a reference icon is drawn in the right bottom corner of the rectangle. Rules are also decorated, i.e., they are represented as green rectangles, and the name of the rules is written inside the rectangle. Figure 10 shows a simple rule with a start symbol, a non-terminal reference, and terminals.

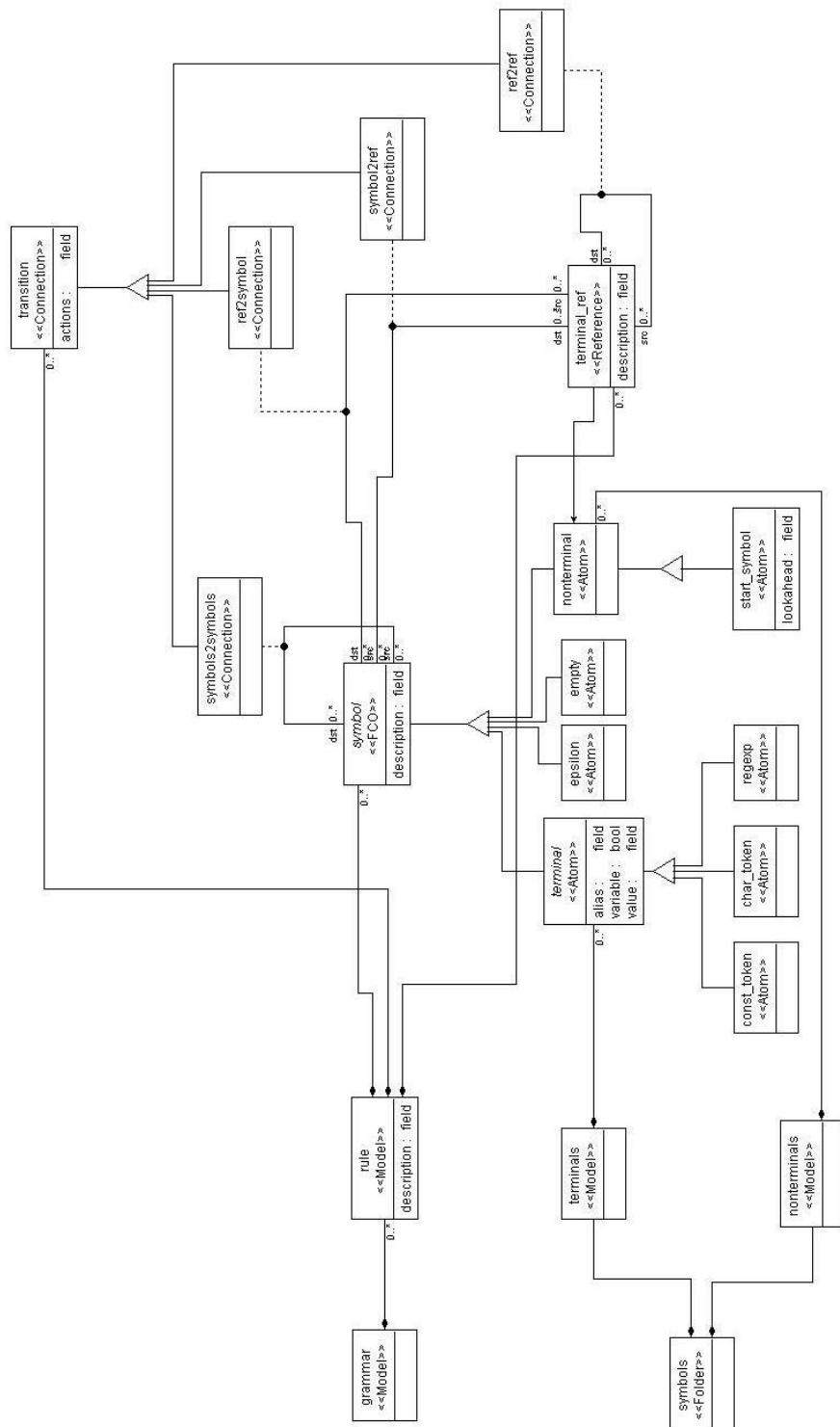


Figure 9 The context-free grammar metamodel

Constraints

The metamodel addresses the goal to make the grammar development and design easier. However, providing a domain-dependent modeling environment is not sufficient because certain construction rules for models in the paradigm cannot be expressed only with class diagrams. For this reason, GME provides a formal definition language to write these *rules* as constraints, and this language is compatible with the OMG standard OCL (Object Constraint Language). The constraints provided by CFG_ML paradigm perform semantic and syntactic checks on the model as follows:

- Constraints on the grammar:
 - One and exactly one start symbol must be defined grammar.
 - A non-terminal symbol instance must appear only on the left-hand side of no more than one rule.
- Constraints on a rule:
 - A rule must have exactly one non-terminal on its left hand side.
 - The name must be valid with respect to constraints on identifiers; i.e., no empty string or white spaces are permitted.
 - The name of the rule must be unique to avoid name collision.
 - The rule must contain only instances of symbols.
- Constraints on a terminal folder:
 - The name of a terminal symbol must be unique to avoid name collision within the grammar.
- Constraints on a non-terminal folder:
 - The name of a terminal symbol must be unique to avoid name collision within the grammar.
- Constraints on a terminal symbol:
 - Terminal symbols must be in upper case to facilitate readability.
 - The value property of a terminal symbol must be valid, i.e., the property cannot be empty.
- Constraints on a non-terminal symbol:
 - Terminal symbols must be in lower case to facilitate readability.
- Constraints on a non-terminal reference:

- The reference must refer to an existing non-terminal; otherwise, the rule cannot be evaluated.

Most of the constraints are checked on the fly, i.e., whenever a possibly relevant change occurs during editing. However, the user can also initiate constraint checking by choosing the corresponding GME command. The error messages and warnings are descriptive and pinpoint the source of the error.

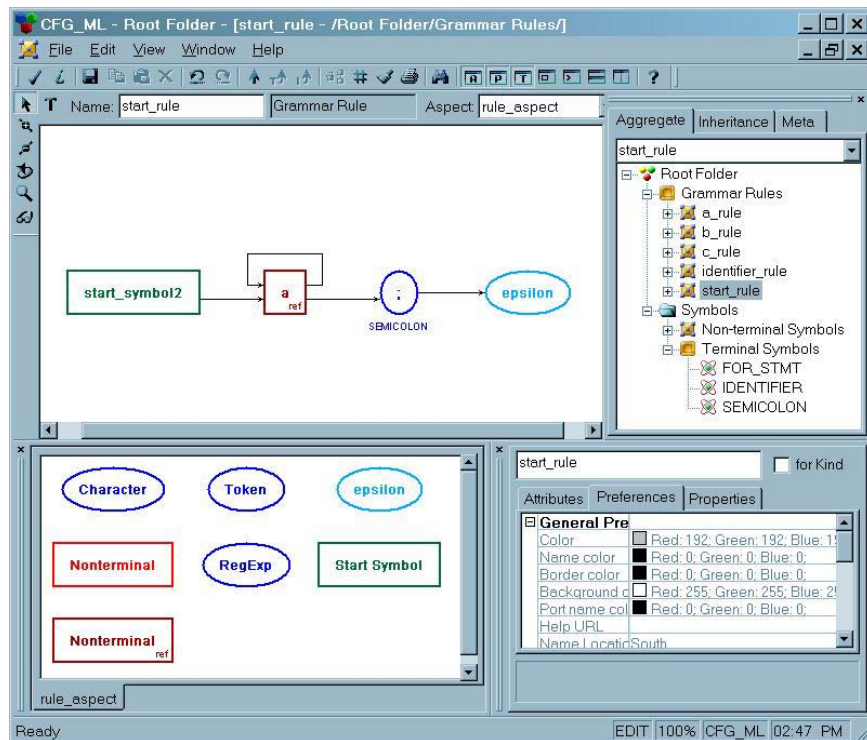


Figure 10 An example rule

Implementing Regular Expressions

The implementation of regular expression does not only process single characters but also includes the following [12]:

- Character Sets: [a-z]
- Negated Character Sets: [^0]
- Any character: .

- One or more operator: +
- Repeat operator: {n[,n]}
- Optional operator: (A)?
- Zero or more operator: *
- The alternation operator: |

The optional operator is basically a specialization of the zero or more operator, and the *any character* (.) meta-character is a special case. Regular expression examples [12]:

- [0-9]* : sequence of digits, i.e., an integral value, for example “19”
- 0|[-+]?[1-9][0-9]* : a zero digit or a sequence of non-zero digits with or without the +/- sign, ie., the zero or any other positive or negative integral value, for example “-138”
- [1-9][0-9]{0,2}([0-9]{3})* : for example “28” or “282293933”
- ([-+]?(((0-9)+[.]?)(0-9)*[.](0-9)+)([eE][-+]?[0-9]+)?

UDM object construction

The language of UDM construction actions should be as simply as possible to avoid complexity; for that reason, the metamodel supports three types of UDM object construction actions: object creation, attribute setting, and association definition. These actions are interpreted whenever the parser finds them during the parsing process. The actions are associated with connections between symbols; as a result, we can assign actions to a symbol not only when entering but also when leaving one.

The UDM object construction actions facilitate object network building in a powerful, convenient, and flexible way. The general syntax of the actions is:

```
command:param1:param2:param3:param4
```

, where `command` specifies the type of the action, such as object and association creation and attribute setting of the object. The rest of the action contains the parameters of the action as described below, and these parameters are separated by colons (:) from the command.

Terminal symbols represent tokens in the input text; for that reason, these terminal symbols can be used as built-in variables. The parser stores these variables as key-value pairs, where the key is the *alias* property of the terminal symbol, and the value is the token the terminal symbol represents. The *variable* property of the terminal symbols is must be set to *true*;

otherwise, the parser omits the symbol as a variable. However, we must be careful because we can not only initialize but also overwrite the value of a given this variable. For example, assume that the *alias* property of a terminal symbol is set to *p1*, and the token this symbol represents is “*trigger*”. If the *variable* property of the terminal symbols is set to *true*, the parser creates a string key-value pair and stores “*p1*” as the key and “*trigger*” as the value of this pair. From now on, we can use “*p1*” in the parameter list of any commands:

```
command:p1:param2:param3:param4
```

The parser substitutes “*p1*” with its value, so the parser interprets the following command whenever encountering the above command:

```
command:trigger:param2:param3:param4
```

A UDM network requires the existence of a root object that serves as a root container of the network. For that reason, the grammar provides a built-in variable called `root` to identify this root object, and we can use this identifier in the parameter list of any commands:

```
command:root:param2:param3:param4
```

The parser uses the root object of the object network in the command above.

Object creation

The object creation action is the most important for the object network. It can be used to create an object with a given name, with a given type, and within a given parent object. The syntax for the object creation action is:

```
create:obj:obj_type:obj_parent
```

where `obj` specifies the variable name of the object to be created, `obj_type` specifies the type of the object to be created, and `obj_parent` specifies the parent object in which `obj` is created.

Figure 11 shows a small class diagram that can help to demonstrate how object creation works.

The following actions result in the creation of two objects:

```
create:task1:Task:root
```

```
create:subTask1:SubTask:task1
```

The first object is the child of the root object of the network, `Container`, its type is `Task`, and we can use `task1` as a variable to refer to this created object later. The second object is a

child of the previously created task `task1`, its type is `SubTask`, and we can also use `subTask1` as a variable to refer to this created object later.

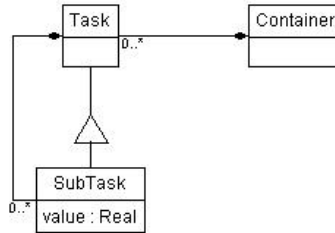


Figure 11 Object hierarchy in a class diagram

Attribute setting

The attribute setting action is used when certain property of an already created object must be set. The syntax for the object creation action is:

```
set:obj:attribute_name:variable_type:variable_value
```

where `obj` specifies the object whose attribute must be set, `attribute_name` specifies the name of the variable that must be set, `variable_type` specifies the type of the variable that must be set, and finally `variable_value` specifies the new value for the variable.

UDM, and the thus the attribute setting actions, provides four types of attributes, namely boolean, integral, real, and string types. The small class diagram on Figure 11 is also can be used to demonstrate how attribute setting works.

Example:

```

create:task1:Task:root
create:subTask1:SubTask:task1
set:subTask1:value:real:val

```

As a result of these two object creation and one attribute setting actions, a small object network will be created with two objects; furthermore, the `value` attribute of `subTask1` will be set to `val`. The parameter `val` refers to a built-in variable; in other words, there is a preceding terminal symbol whose alias is `val`, and the token it represents will be assigned to the `value` attribute.

The attribute setting action above shows that `subTask1` has a real attribute `value`; however, we can use other types to set the value of an attribute as follows:


```

set:obj1:init:int:val
set:obj2:init:real:val
set:obj3:init:string:val
set:obj4:init:bool:val

```

The attribute setting actions above show that `obj1` has an integer attribute `init`, `obj2` has a real attribute `init`, `obj3` has a string attribute `init`, and `obj4` has a boolean attribute `init`.

Associations

The most important feature of UDM is the capability to associate, i.e., to connect object in the object network. For that reason, this grammar provides a simple action that can be used to associate certain objects via a given role. The syntax for this action is:

```

assoc:obj:role_name:assoc_obj

```

where `obj` specifies one end of the association that has the role `role_name`, and `assoc_obj` specifies the object to be associated with `obj`.

Figure 12 shows a small class diagram that demonstrates how to define a simple association between two objects:

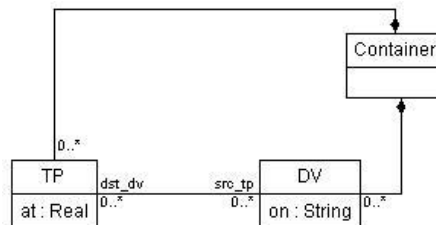


Figure 12 Simple association in a class diagram

The following actions result in the creation of two objects and an association for the class diagram showed on Figure 12:

```

create:dv_obj:DV:root
create:tp_obj:TP:root
assoc:dv_obj:src_dv:tp_obj

```

The first object, `dv_obj`, is the child of the root object of the network, `Container`, and its type is `DV`. The second object, `tp_obj`, is also a child of the root object, and its type is `TP`. The association action connects the object `tp_obj` to the object `dv_obj` via the given role name `src_tp` of `dv_obj`. Note that the references `dv_obj` and `tp_obj` are used to identify existing objects in the network.

However, sometimes there is a need to assign attributes to association. For that reason, an association class is used to define a connection between two objects as shown on Figure 13.

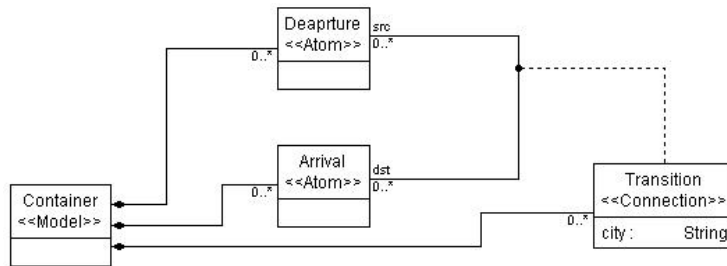


Figure 13 Association class in a class diagram

The following object construction actions create a small object network with two objects, one association class, a connection between the two objects via the association class, and sets the attribute `city` of the connection:

```

create:a:Arrival:root
create:d:Deaprture:root
create:tr:Transition:root
assoc:a:src:tr
assoc:d:dst:tr
set:tr:city:string:city_id

```

The created object network is shown on Figure 14.



Figure 14 Association via association class

CHAPTER IV

IV. INTERPRETIVE PARSING

Any text can be generated from a UDM network by using the UdmOclPat program that interprets UDM data network and generates text output through a simple pattern based query language [6]. This thesis addresses the goal to close the loop, i.e., to generate a UDM network by parsing a text file. For that reason, a parser tool is needed, and the parser is used to build up a UDM network from the text file based on a grammar. However, this interpretive parser does not construct a parse tree; rather, it constructs a UDM network as the result of object construction actions. This chapter focuses on a parser tool that provides convenient interfaces to parse texts and build up a UDM network based on the grammar we created with the grammar modeling tool.

Introduction

A new grammar parser is developed for UDM in this thesis. This application is capable of generating a UDM data network given any text file and a grammar file. The grammar file contains the grammar rules of parsing and the UDM object construction actions. The text file contains the arbitrary text that the parser can parse and analyze with the help of the UDM object construction actions.

The typical usage of the interpretive parser corresponds to Figure 2 on Page 6. The interpretive parsing tool takes three inputs, and processes them to build an object network. The first input is a text file that contains arbitrary text to be parsed. The second input is the data network the parser modifies by interpreting UDM object construction actions. The third and final input is the grammar file that contains not only the parsing rules but also the construction actions that the parser interprets.

The interpretive parser is a library, and the parsing is available as a library function. The following example shows how the library function is invoked:

```
Text2UdmParser::Parser parser(<text file>, <data network>, <grammar
file>);
parser.parse();
```

This function is in the Text2Udm.lib library, which is the main objective of my thesis. As previously introduced, the first parameter is text file with arbitrary text to be parsed, the second parameter is the data network to be modified, and the third parameter is the grammar file. The parser can throw two kinds of exceptions while running, so it is necessary to handle the `ParserException` and the `udm_exception` it might throw.

The parser tool provides logging capabilities so that the user can decide whether to log messages in a file. Therefore, the fourth parameter is optional, and by default the parser places a log file name `text2udm.log` and writes information and error messages in this file. As a result, the library function can also be invoked as follows.

```
Text2UdmParser::Parser parser(<text file>, <data network>, <grammar
file>, <logger parameter>);
parser.parse();
```

The 3rd party library `Logger` [20] is used to facilitate logging. This library is distributed under the terms of GNU General Public License [21]. The library has the advantage that it is easy to configure and several levels of logging are defined; however, it is required to add the `logger.dll` 3rd party library to the `PATH` environment variable. There are four levels of logging supported in the parser tool as follows:

- `Parser::NO_LOG`: no message is directed to the log file
- `Parser::LOG_INFO`: only information messages are directed into the log file
- `Parser::LOG_ERROR`: only error messages are directed into the log file
- `Parser::LOG_INFO_ERROR`: both information and error messages are directed into the log file

For example, if we want to invoke the library function so that only the error messages appear in the log file, we have to change the fourth parameter as follows:

```
Text2UdmParser::Parser parser(<text file>, <data network>, <grammar
file>, Parser::LOG_ERROR);
parser.parse();
```

Turning on the logger is suggested when using the interpretive parser tool for the first time.

Internal Operation of the Interpretive Parser Tool

The parser performs five steps when processing its input and another step when building the object network as shown on Figure 15.

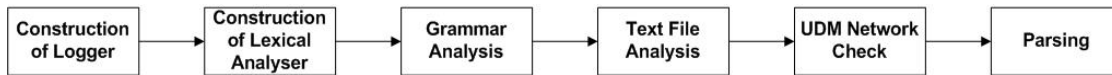


Figure 15 The six steps performed by the parser

First, the parser constructs and configures the logger that will be used during parsing. Second, it constructs a lexical analyzer that will recognize tokens from the input text file. Third, it performs some basic analysis on the given, such as detecting left-recursion or other ambiguity or constructing parsing table. During the fourth step, the parser analyses the text file; in other words, it checks whether the text file exists and opens it for reading. In the final step of the first phase, the parser checks whether the given data network is valid. In other words, it tests whether the class diagram from which the data network should be designed exists. In the second phase, the parser not only parses the text file against the grammar rules but also interprets the built-in UDM object construction actions of the grammar.

Logging

The logger needs write permission for the directory where the parser library is used because it tries to create a log file. If the user does not have write permission, the parser cannot perform the next step.

Lexical Analysis

The interpretive parser library *text2udm* uses the 3rd party library BHLex [12] for lexical analysis of the input text. Although this library is free, it is not under the terms of GNU General Public License [21]. The library also has the advantage that it is easy to configure; however, this lexer library is statically linked to the parser library, so no 3rd party library is needed. The main advantage of BHLex is that it does not generate source code but constructs a state machine at

runtime; thus, a library function must be invoked to start the lexical analysis of the text. The lexer recognizes the following tokens:

- Keywords: sequence of either upper case or lower case letters
- Key characters: key characters, such as (,), ' ', ., ;, /, -, +
- Identifiers: C-style identifier, i.e., a string started with a character or underscore followed by digits, characters, or numbers
- Numbers: sequence of digits
- Other tokens: any combination of digits and characters

The lexer recognizes whitespaces and ignores them. As usual, keywords have preference over identifiers; identifiers have precedence over numbers, and so on.

Grammar Analysis

One of the inputs of the parser is a context-free grammar model that uses the context-free grammar metamodel developed in the first part of the thesis. The interpretive parser library uses a grammar library to perform the parsing, and this grammar library is generated by UDM. In other words, the parser does not construct a parse tree, it uses the UDM object network correspondence of the original input context-free grammar model when parsing. The grammar library is statically linked to the parser library.

When the parser performs the grammar analysis, then it basically performs sanity checks. First, it checks whether a start symbol is defined for the grammar. This start symbol is a distinguished non-terminal symbol, and the set of strings it denotes is the language defined by the grammar. If no start symbol is defined, then the parser cannot find the entry point of the grammar rules; as a result, parsing cannot be completed.

Second, the parser tries to detect indirect and immediate left-recursion. Since the parser is a recursive-descent one, it is possible for it to loop forever when the leftmost symbol of the right side of the rule is the same as the non-terminal on the left side. For that reason, it is essential to detect left-recursion before starting to parse the text to avoid abnormal program termination.

In the next step, the parser checks whether the grammar is LL(1) because the parser uses a single lookahead when making decisions. If the productions of a rule have a common prefix, the parser could not determine which production was going to successfully match; hence, the rule is ambiguous in the LL(1) sense.

The parser is table-driven; in other words, a table is used to guide the parser when making decisions. This table is computed by the parser by using the first sets of the non-terminal symbols, and this table can guide a parser of any language which has an LL(1) grammar. The first set of a non-terminal symbol contains the set of terminals that begin the strings derived from the given non-terminal [14]. Thus, in the final step, the parser constructs the first tables for its non-terminal symbols that will be used when making parsing decisions.

If any of these steps fail, the library throws exceptions with descriptive messages and returns to its invoker.

Text file Analysis

The parser receives a text file as an input, and it needs to perform some basic checks to avoid I/O errors during parsing. In case the text file does not exist or cannot be read, i.e., the file cannot be opened, the parser cannot perform the next step and throws an exception that explains the problem.

UDM Object Network

The parser needs a UDM data object network to perform the UDM object construction actions. It is not a requirement to give a new or empty network to the parser because the parser does not take whole network into consideration and does not delete objects; on the contrary, it constructs objects and modifies these constructed objects. However, it is a requirement to give an opened object network with UML class diagram to the parser. It is not necessary to explicitly pass the UML class diagram, but the object network must be opened; otherwise, the parser throws an exception explaining the problem and exists.

Interpretive Parsing

The parsing process mimics the recursive-descent parsing technique combined with predictive parsing. In other words, the parser does not contain a procedure for every non-terminal symbol but contains only one procedure that is used recursively for every symbol. This procedure does exactly two things: it decides which production to use by looking at the lookahead symbol and uses the chosen production to invoke the recursive function and mimic the

right side. If there is a conflict between the two right sides for any lookahead, then we cannot use this parser on the given grammar.

The pseudo-code description of the parser algorithm is shown in the Appendix.

The parser starts the parsing from the start symbol of the grammar. If the parser encounters a non-terminal symbol, it makes its parsing decision by using a single lookahead token. When the design decision is successfully made and the next symbol is found, the parser checks the connection between the non-terminal and the next symbol and interprets the UDM object construction actions if there is any. Similarly, whenever the parser encounters a reference to a non-terminal symbol, it resolves the reference to a non-terminal symbol and recursively calls the parse function by passing the resolved non-terminal symbol. When the recursion returns, the parser advances to the next symbol of the production, interprets the UDM object construction actions if there is any, and calls itself recursively by passing the next symbol of the production. A terminal symbol results in matching the token with the lookahead, and the next input token will be read. If the terminal does not match the lookahead, an error is declared. Again, the parser moves to the next symbol of the production, interprets the UDM object construction actions if there is any, and calls the recursion by passing the next symbol of the production.

When interpreting the UDM object construction actions, the parser uses two tables for resolving variable names in the actions: one for the terminal symbols and one for the UDM objects. The terminal symbols can be used as variables during the lifetime of the parser providing that their *alias* and *variable* properties are set correctly. The created UDM objects are automatically registered with the given object name in the object variable table, and their lifecycle is similar to the other variable table. The object network specific API is not needed for the parser because the interpreter uses the UDM TOMI Paradigm Independent Interface, a generic interface [4] to perform object construction; however, the generic API is not as efficient as the domain specific, generated one. If the generic interface encounters a problem, the parser forwards the error message to its invoker and exists.

CHAPTER V

V. EXAMPLES AND EVALUATION

This chapter explains the usage and functioning of the language parser using some examples. A step-by-step approach is taken to explain the construction of a grammar file to achieve a desired output.

State Chart

In this first example, we look at a text that describes some state chart, and we use the State Chart GME paradigm [5] to build an object network as shown on Figure 16.

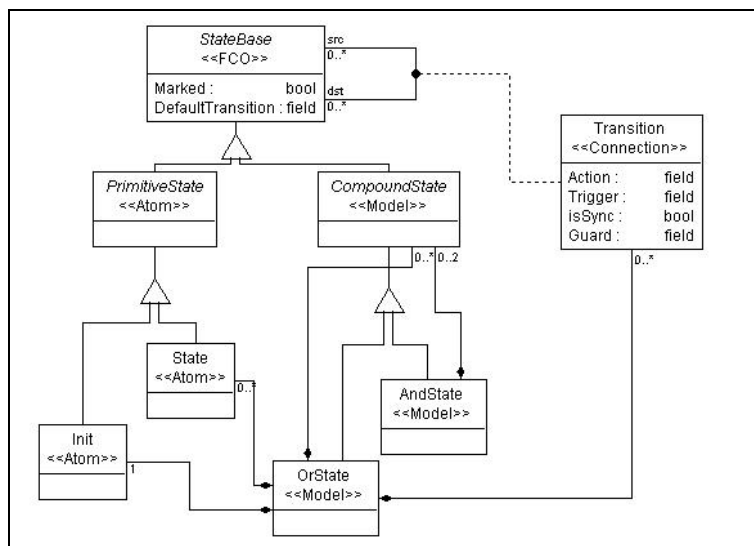


Figure 16 GME metamodel for the State Chart paradigm

In the State Chart paradigm simple states, such as initial state and primitive state, and compound states, such as and OR or AND state can be created. The paradigm also defines transitions between these states, and transitions have attributes for setting triggers, guards, and actions. A simple state chart model is shown on Figure 17 containing some states and transitions between them.

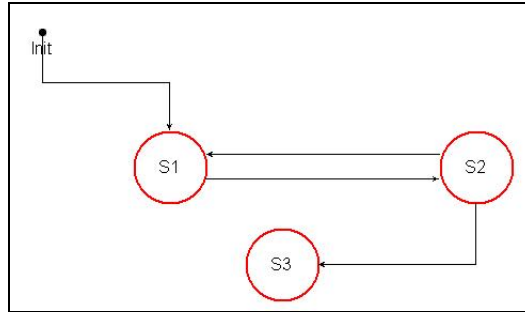


Figure 17 Example model for the State Chart paradigm

The goal is to generate a Start Chart model from a text file. A generic tool is needed that can take the specific text file, parses it, and creates the model. The text2udm library tool meets these requirements.

The following example text describes a state chart with six states:

```

root_or_state NewOrState;           // create a root OR state
and_state (AndState, NewOrState);   // create an AND state
or_state (2_0, AndState);           // create an OR state
init_state (Init, 2_0);             // create an initial state
state (L0, 2_0);                     // create a simple state
state (L1, 2_0);                     // create a simple state
transition (tr2, 2_0);               // create a transition
connect_src (L0, tr2);               // connect states L0 and L1
connect_dst (L1, tr2);               // via transition tr2
transition (tr3, 2_0);               // create a transition
trigger (tr3, x_tr)                  // set the trigger property
connect_src (L1, tr3);               // connect states L0 and L1
connect_dst (L0, tr3);               // via transition tr3

```

The first step is to analyze and understand the text. First, there is a main OR state whose name is `NewOrState` that contains all other states. Then, there is an AND state whose name is `AndState` in the main OR state, and there one initial state and two simple states in the AND state. The two simple states are connected by transitions, and the trigger property of the second transition is set to `x_tr`. The comments in the text file are not part of the text.

The State Chart paradigm tells us how to construct the grammar. First, that the root folder of the generated model must be an OR state. Furthermore, each line of the text corresponds to a command, such as create a state or transition, connect states, and set a property of a state or transition, and each line is closed by a semicolon. The parameters of the state creation actions are: the state to be created and the parent state of this stat, and these parameters are enclosed in parentheses. Similarly, the parameters of the transition creation are the transition to be created and the parent state of transition, and these parameter are also enclosed in parentheses. The connect line has also two parameters, namely the state and the transition, and these parameter are also enclosed in brackets. Setting an attribute of a state or transition starts with the name of the attribute and requires two parameters: the state or transition whose property should be changed and the value to be assigned to the attribute.

Before constructing the grammar, we must also consider the UML metamodel of the State Chart paradigm because we will need this class diagram to build the object network later. This class diagram contains the class name of the elements of the State Chart paradigm, such as the states and transitions, and it also provides important information to facilitate the construction of actions in the grammar. The UML metamodel of the State Chart paradigm is shown on Figure 18. The only difference we can see so far is the change in the role names for the states.

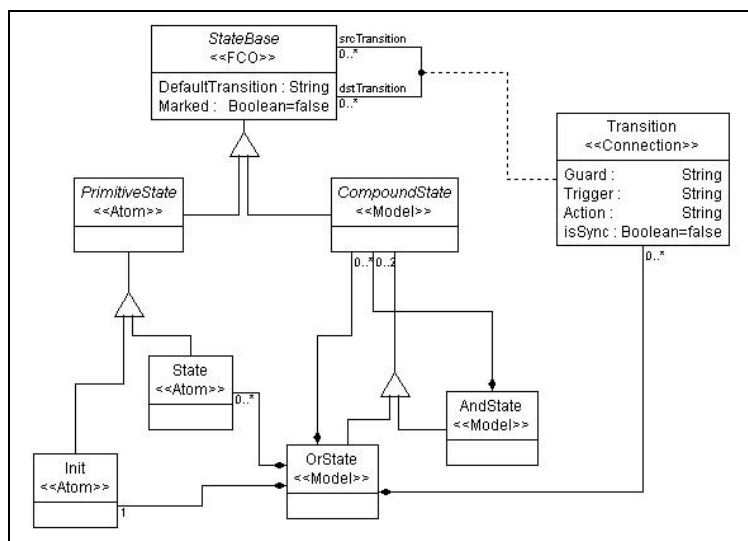


Figure 18 The UML metamodel of the State Chart paradigm

The constructed grammar can be rewritten in EBNF-like format, where the terminal symbols are in between quotation marks:

```
start_symbol := root_or_states elements
root_or_states := (root_or_state)*
root_or_state := "root_or_state" identifier ";"
elements := (element)*
element := and_state | or_state | init_state | primitive_state |
          state_attr | connect | transition | transition_attr
and_state := "and_state" data_pair ";"
or_state := "or_state" data_pair ";"
init_state := "init_state" data_pair ";"
primitive_state := "state" data_pair ";"
state_attr := "default_tr" data_pair ";" | "marked" data_pair ";"
connect := "connect_src" data_pair ";" | "connect_dst" data_pair ";"
transition := "transition" data_pair ";"
transition_attr := "trigger" data_pair ";" | "guard" data_pair ";" |
                  "action" data_pair ";" | "sync" data_pair ";"
data_pair := "(" identifier "," identifier ")"
identifier := [a-z|A-Z|0-9|_]+
```

The corresponding context-free grammar with the UDM construction actions can be found in the Appendix.

We have the text file and the grammar so far, so we can use the text2udm library to generate the state chart from the text as follows:

```

// create UDM data newtork
Udm::SmartDataNetwork out(StateChart::diagram);
out.CreateNew("output.xml", "StateChart.xsd",
    StateChart::RootFolder::meta, Udm::CHANGES_PERSIST_ALWAYS);
Udm::Object o_root = out.GetRootObject();

// create the parser
Text2Udm::Parser parser("sc_text.txt", &out, "SC_grammar.mga");
parser.parse();

// create GME model
Udm::SmartDataNetwork out2(StateChart::diagram);
out2.CreateNew("output.mga", "StateChart",
    StateChart::RootFolder::meta, Udm::CHANGES_PERSIST_ALWAYS);
out2 = out;

// close data networks
out.CloseWithUpdate();
out2.CloseWithUpdate();

```

The C++ source code above will create and open a UDM object network with XML backend and a UDM object network with GME backend; thus not only an XML file will be generated but also a GME file that could be opened and checked. The outputs of the execution of the source code above can be found in the Appendix.

Time Series

A time series is a collection of ordered pairs $(t, X(t))$, where t = time and $X(t)$ = value of a signal at a time t . This is the typical output from some kind of biocomputational simulation run, or it is an experimental data obtained as time series of species concentrations (or activities). [22] Time series data is widely used in BioSpice applications [23]. Although a recommendation exists for the format of time series in BioSpice, the applications often do not use it.

A group of time series that belong together is identified by a label, and this label is associated with its corresponding time and data values and stored in the root container of the

network. The users of time series usually want to access directly not only the labels of the time series but also the time and data values; for that reason, all the time and data value should be stored in the root container of network, too. The UML metamodel for time series that meets the criteria described above is shown on Figure 19.

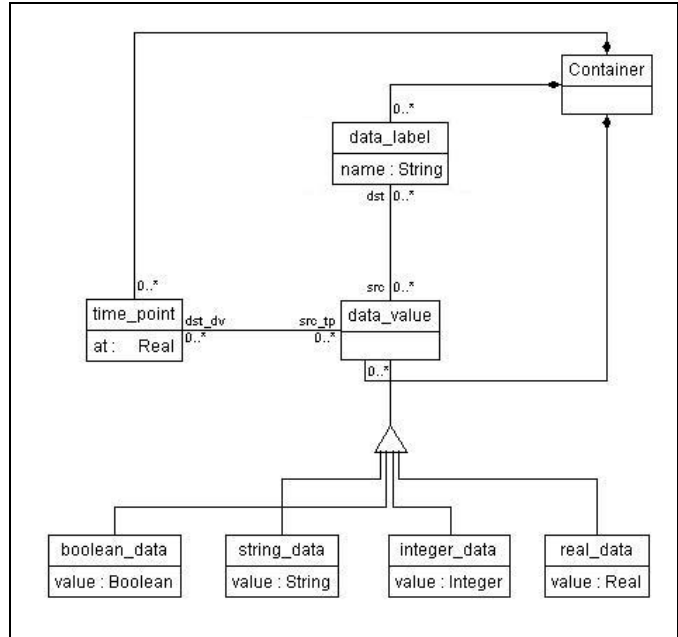


Figure 19 UML metamodel for Time Series

The container of time series contains labels, data values, and time points. Each data value is associated with one time point and a time series label, and the type of data value can be boolean, integral, real, and string.

A common format for time series contains series of labels and data pairs and uses real values as follows [22]:

```

(
Cdc25_activation_Ca
(1.25, 0.8)
(2.5, 0.9)
(5, 1.0)
(10, 1.0)
)
(
Cdc25_inactivation_Ca
(5, 0.75)
(10, 0.5)
(20, 0.1)
(40, 0.0)
)

```

The goal is to generate a UDM object network from a text file. Again, a generic tool is needed that can take the specific text file, parses it, and creates the model. The text2udm library tool meets these requirements.

The first step is to construct the grammar that describes the text file above. Each time series is enclosed between brackets, and each time series contain a label that identifies the time series and a list of time – data value pair enclosed in brackets. The constructed grammar can be rewritten in EBNF-like format, where the terminal symbols are in between quotation marks:

```

start_symbol := series_container_list
series_container_list := (series_container)*
series_container := "(" data_label series_list ")"
data_label := [a-z|A-Z|0-9|_]+
series_list := (data_pair)*
data_pair := "(" time_point "," data_value ")"
time_point := [0-9]+[.]?[0-9]*
data_value := [0-9]+[.]?[0-9]*

```

The corresponding context-free grammar model with UDM construction actions can be found in the Appendix.

However, there is another common format for time series that contains series of triplets: a label that identifies the timer series, the time point, and that data of real and boolean values [22]:

```
(
(Cdc25_activation_Ca, 1.25, 0.8)
(Cdc25_activation_Ca, 2.5, 0.9)
(Cdc25_activation_Ca, 5.0, 1.0)
(Cdc25_activation_Ca, 10, 1.0)
)
(
(Cdc25_inactivation_Ca, 5.0, true)
(Cdc25_inactivation_Ca, 10, 0.5)
(Cdc25_inactivation_Ca, 20, 0.1)
(Cdc25_inactivation_Ca, 40, 0.0)
)
```

By slightly modifying the grammar above, we can construct a grammar for this format. The constructed grammar can be rewritten in EBNF-like format, where the terminal symbols are in between quotation marks:

```
start_symbol := series_container_list
series_container_list := (series_container)*
series_container := "(" series_list ")"
series_list := (data_triplet)*
data_triplet := "(" data_label series_list "," time_point ","
    data_value ")"
data_label := [a-z|A-Z|0-9|_]+
time_point := [0-9]+[.]?[0-9]*
data_value := [0-9]+[.]?[0-9]* | boolean
boolean := "true" | "false"
```

The corresponding context-free grammar model with UDM construction actions can be found in the Appendix.

We have the text file and the grammar so far, so we can use the text2udm library to generate the object networks from the texts:

```
// create data network
Udm::SmartDataNetwork out(TimeSeries::diagram);
out.CreateNew("output.xml", "TimeSeries.xsd",
             TimeSeries::Container::meta, Udm::CHANGES_PERSIST_ALWAYS);
Udm::Object o_root = out.GetRootObject();

// create the parser
Text2Udm::Parser parser("time_series_text.txt", &out, "TS_grammar.mga");
parser.parse();

// close data network
out.CloseWithUpdate();

// create data network
Udm::SmartDataNetwork out2(TimeSeries::diagram);
out2.CreateNew("output_triplet.xml", "TimeSeries.xsd",
              TimeSeries::Container::meta, Udm::CHANGES_PERSIST_ALWAYS);
Udm::Object o_root2 = out2.GetRootObject();

// create the parser
Text2Udm::Parser parser2("time_series_triplet_text.txt", &out2,
"TS_grammar_triplet.mga");
parser2.parse();

// close data network
out2.CloseWithUpdate();
```

The C++ source code above will create and open two UDM object networks with XML backend, and builds up an object network for the simple time series and the time series with triplets. The outputs of the execution of the source code above can be found in the Appendix. Notice, that we use the same UML metamodel to build the object networks by using different text formats and different grammars; moreover, the two data networks are completely identical.

Evaluation

In this section, we will compare the generation of object networks from text files using the provided text2udm library with another alternative techniques.

One use case is when an object network is in some kind of text format, and one wants to use that with UDM. One alternative is to generate the UDM framework from the UML metamodel of the given UDM network, and write a short program in which object creation commands appear in the same order objects appear in the text file. The drawback of this method is that whenever a new text file of different format is encountered, a completely new application must be written, and the source code of any other application cannot be entirely reused. In addition, the source code has to be modified if the same text file is handled slightly differently, and the application also has to be recompiled, re-linked, and re-executed. Thus, this method is basically a write once – use once technique. However, the provided text2udm library can automate the process of generating object networks from text files. The provided grammar modeling tool can be used to define the UDM object construction actions, and only this grammar file must be modified whenever a new text of different format is encountered. There is no need of re-compilation or re-link of any source code; you only have to change the parameters when invoking the text2udm library function.

Another use case is when the object networks are in some kind of text format, and a visual representation of that text by using a GME paradigm should be build. GME provides an interface that facilitates object network creation from XML, but does not provide an interface that facilitates object network creation from arbitrary text. For that reason, the GME tool would be used to create and connect the objects by hand. This is a really tedious and cumbersome work for larger networks and large number of text files. However, UDM provides a GME backend, so the text2udm library function can be used once the grammar for a family of text files is constructed. The provided library will generate GME models with the same paradigm from any text files that can be parsed with the given grammar.

Although the parser library can be used successfully to build object networks from simple text files, it has certain limitations. First, the parser cannot parse text files whose language cannot be described with LL(1) grammars. In addition, the parser does not provide strategies to recover from errors, so it exits rather than recovering and continuing parsing. Furthermore, the built-in lexer is configured statically rather than dynamically, so the user of the library cannot

control the lexical analysis of the input text. Finally, the developed parser library is built with Visual Studio 6.0; as a result, this library cannot be used neither with Visual Studio .NET nor under UNIX operating systems.

CHAPTER VI

VI. CONCLUSIONS AND FUTURE WORK

Conclusions

Model based programming is becoming increasingly popular and important because it can be used by not only programmers but also people who have specific domain knowledge but little programming knowledge [6]. Many people encounter the problem that the old domain specific data elements are stored in some kind of text format, and modeling tools do not provide interfaces to import these data. In fact, there is a need to be able to import the existing data into the modeling tool in an easy and comprehensive way to avoid loss of data or cumbersome manual transformation of data from one application to the other.

The developed interpretive parser tool provides a simple interface to users for generating object networks from text files. It is generic and can be used with any kind of text providing that the text can be parsed with the interpretive parser. The parser needs a grammar to successfully parse the text and interpret the UDM object construction actions, and creating a grammar with the provided context-free grammar builder tool is also easy and comprehensive. Although the language of the UDM object construction actions is easy to understand and use, writing the UDM object construction actions requires the knowledge of the UML metamodel of the destination UDM object network.

Thus, the context-free grammar tool and interpretive parser tool developed in this project offer several advantages in accomplishing the task of generating object networks from text files.

Future Work

Although the context-free grammar tool and interpretive parser tool can be used to generate object networks from text files, and examples show how to apply them in real life, there are many options for improvements.

The interpretive parser tool simulates a recursive-descent predictive LL(1) parser. However, single lookahead is often not sufficient to make parsing decision, so it could be a research topic how to extend the parsing capability to LL(k).

There are many different general strategies that a parser can employ to recover from a syntactic error and continue parsing instead of exiting with error messages and error codes. Thus,

it could be another interesting research topic to implement a recovery mode and integrate with either the grammar or the parser or both.

The lexical analysis phase of the interpretive parsing can be also improved or evolved. Instead of giving types of tokens to be accepted by the application, it would be more useful for the user if he or she could configure the lexer, too. For that reason, either the ANTLR tool could be used to generate the lexer, or other lexical analyzer tool should be considered.

The text2udm library was developed for Windows OS users; however, there is a growing need to use not only UDM but also this library under UNIX operating systems. To make the text2udm library platform independent or to provide a UNIX executable might require either the re-design of the library or the modification of the source code or both.

A. The pseudo-code description of the parser algorithm

```
tok // global variable to store a token
next() // routine that sets "tok" to token in input
first(symbol s) // the first set of non-terminal symbols s
match(terminal t) // routine that matches the tok with the actual
terminal symbol
interpret(action a) // routine that interprets the action
error() // error handling routine

parse (symbol s)

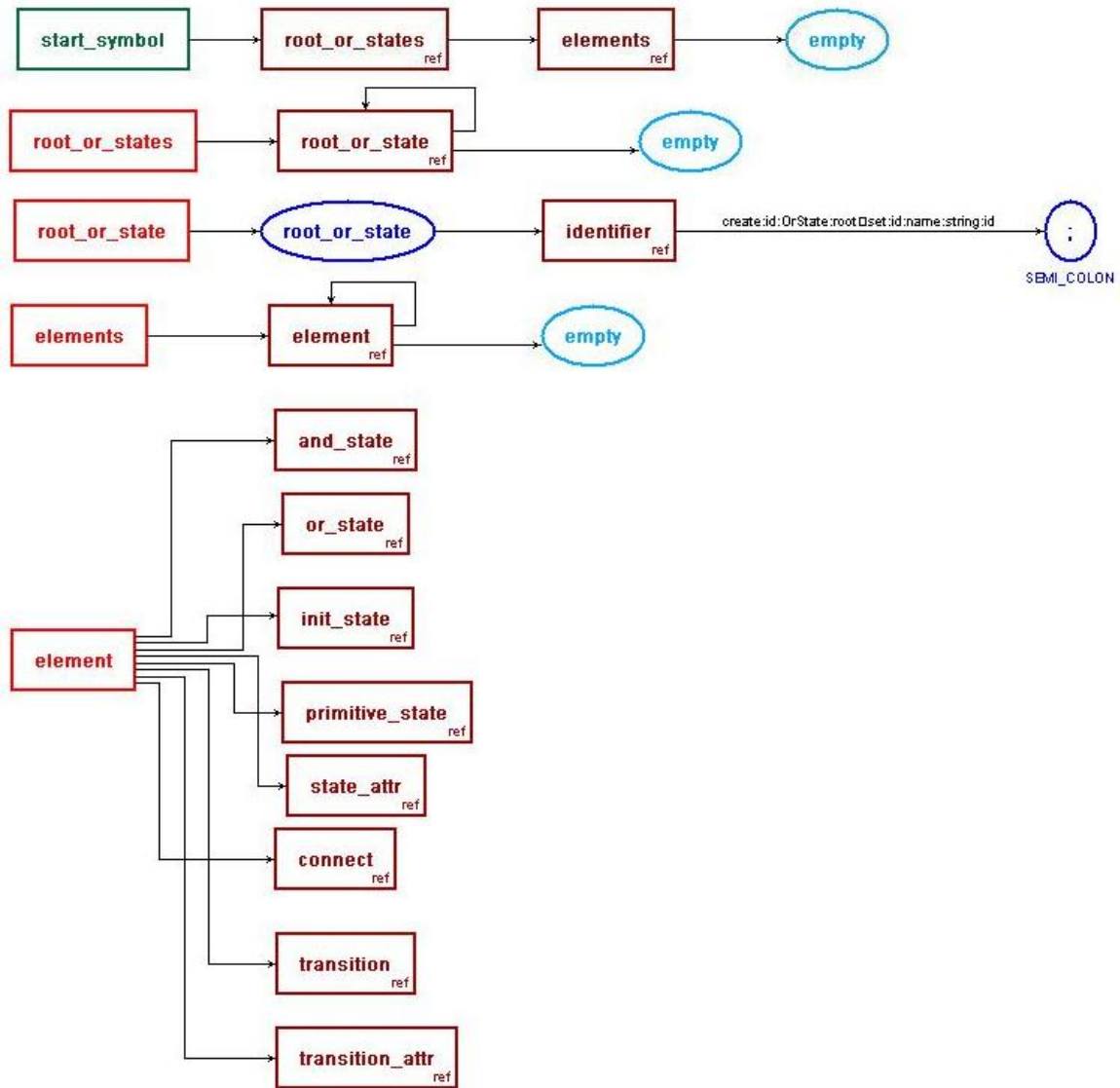
  if (s is nonterminal)
    set productions // the productions of the rule
    forAll (p : productions)
      if (tok is in first(p) OR match(p))
        transition t // transition from s to p
        action a // the object construction actions
        interpret(a)
        parse(p) // call itself recursively
        break

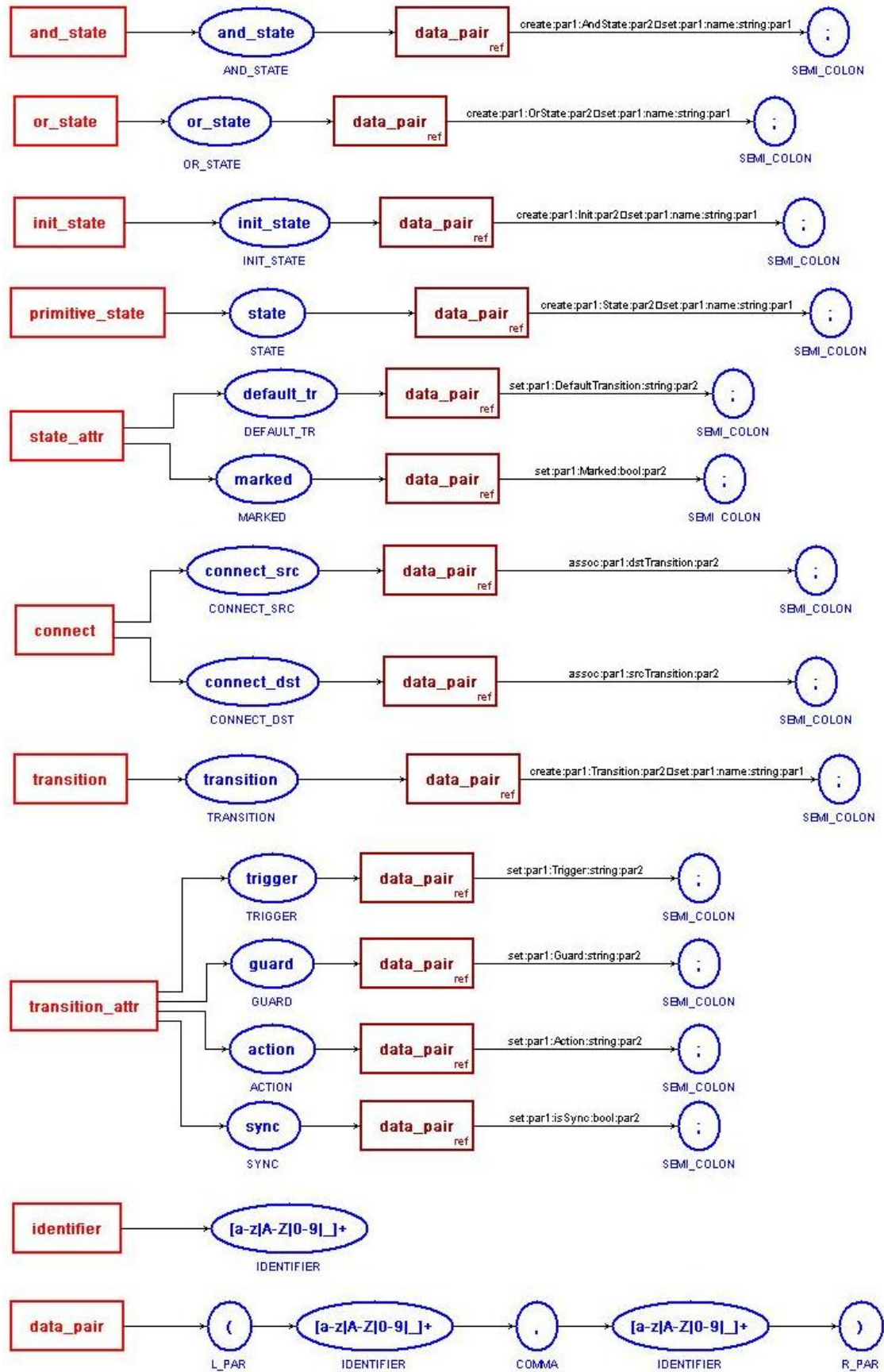
  else if (s is reference to a nonterminal)
    symbol refTo // resolve the reference
    parse (refTo) // call itself recursively
    set symbols // the next symbols of the productions
    forAll (p : symbols)
      if (tok is in first(p) OR match(p))
        transition t // transition from s to p
        action a // the object construction actions
        interpret(a)
        parse(p) // call itself recursively
        break

  else if (s is terminal)
    if (match(s))
      if (s.variable = true)
        variable_table.add(s.alias, tok)
      next()
    else
      error()

  set symbols // the next symbols of the productions
  forAll (p : symbols)
    if (tok is in first(p) OR match(p))
      transition t // transition from s to p
      action a // the object construction actions
      interpret(a)
      parse(p) // call itself recursively
      break
```

B. Grammar description of state chart text

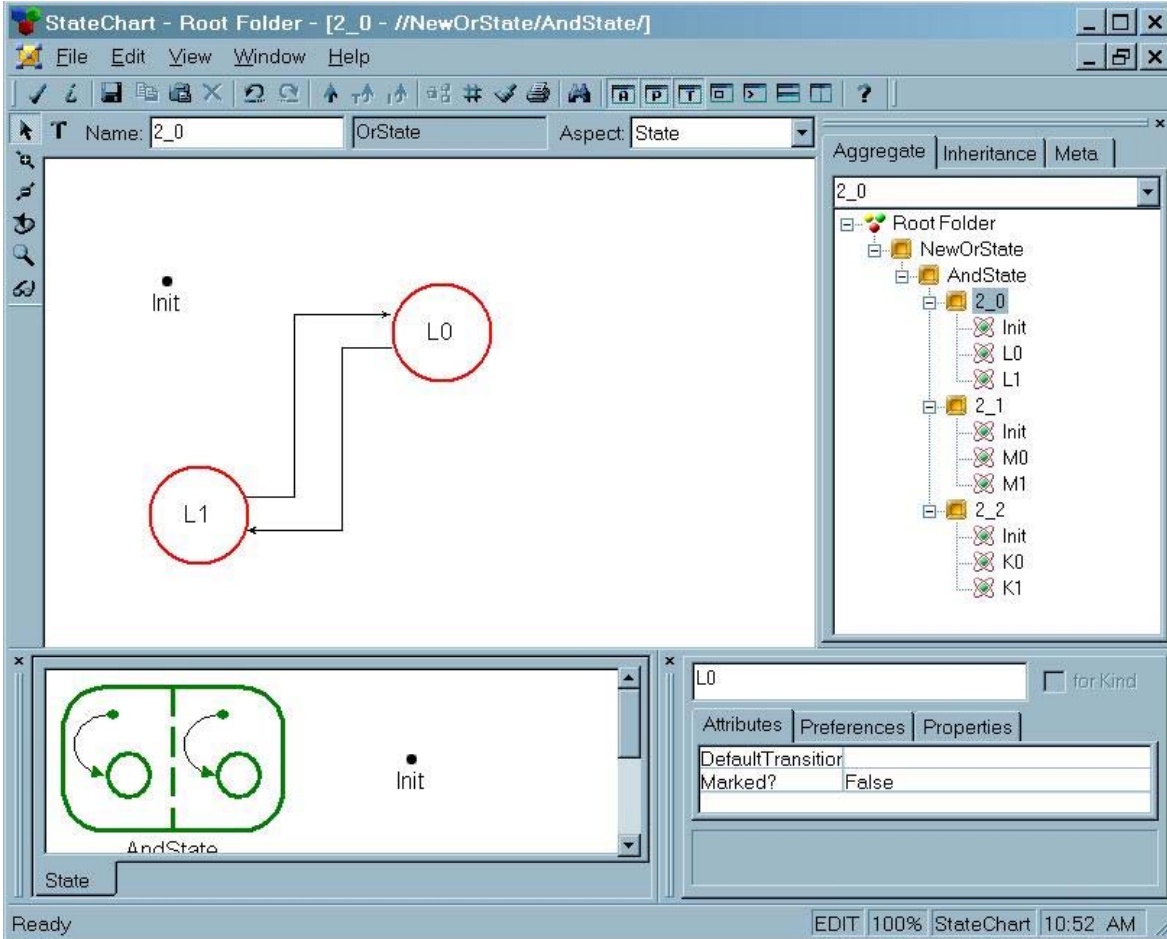




C. The state chart text

```
root_or_state NewOrState;
and_state (AndState, NewOrState);
or_state (2_0, AndState);
init_state (Init, 2_0);
state (L0, 2_0);
state (L1, 2_0);
transition (tr2, 2_0);
connect_src (L0, tr2);
connect_dst (L1, tr2);
transition (tr3, 2_0);
connect_src (L1, tr3);
connect_dst (L0, tr3);
or_state (2_1, AndState);
init_state (Init, 2_1);
state (M0, 2_1);
state (M1, 2_1);
transition (tr4, 2_1);
connect_src (M0, tr4);
connect_dst (M1, tr4);
transition (tr5, 2_1);
connect_src (M1, tr5);
connect_dst (M0, tr5);
or_state (2_2, AndState);
init_state (Init, 2_2);
state (K0, 2_2);
state (K1, 2_2);
transition (tr7, 2_2);
connect_src (K0, tr7);
connect_dst (K1, tr7);
transition (tr8, 2_2);
connect_src (K1, tr8);
connect_dst (K0, tr8);
```

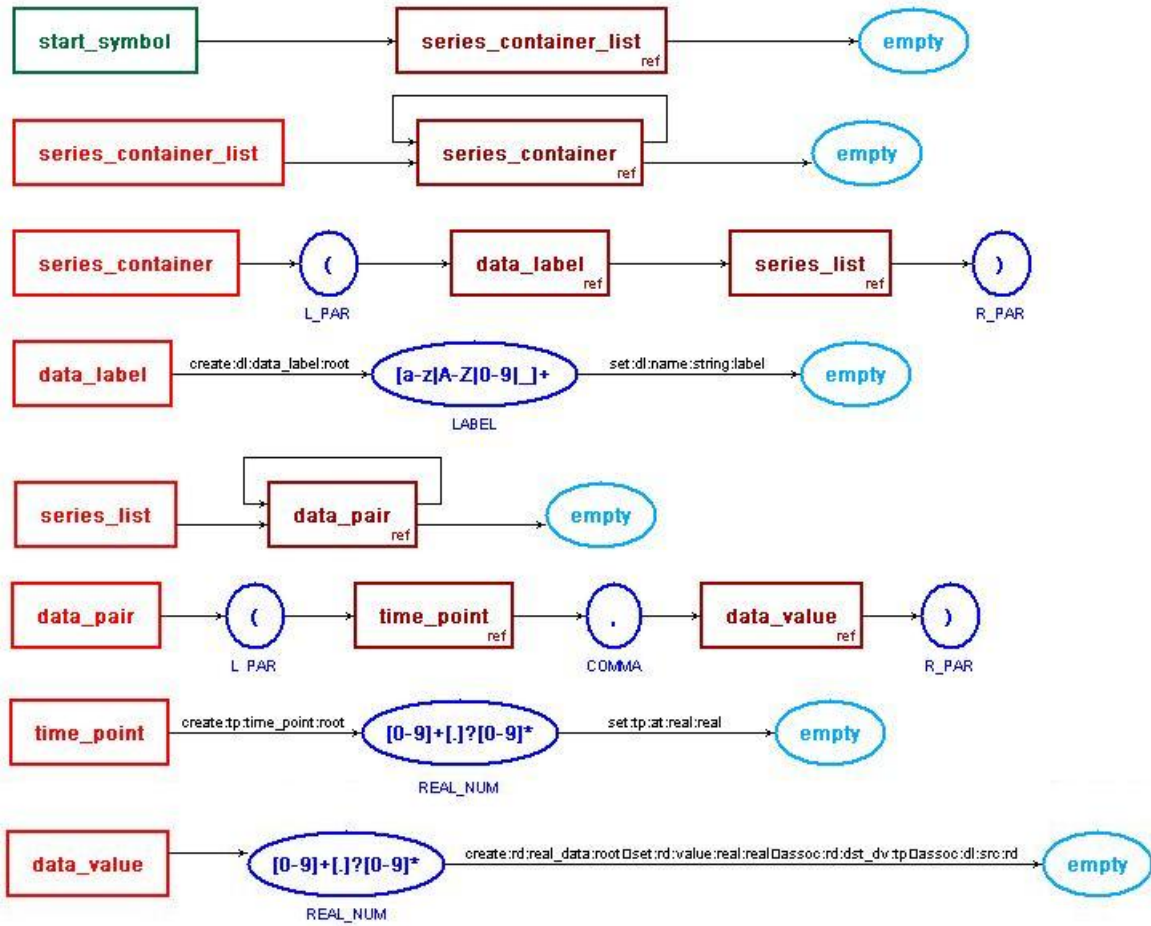
D. The output State Chart model



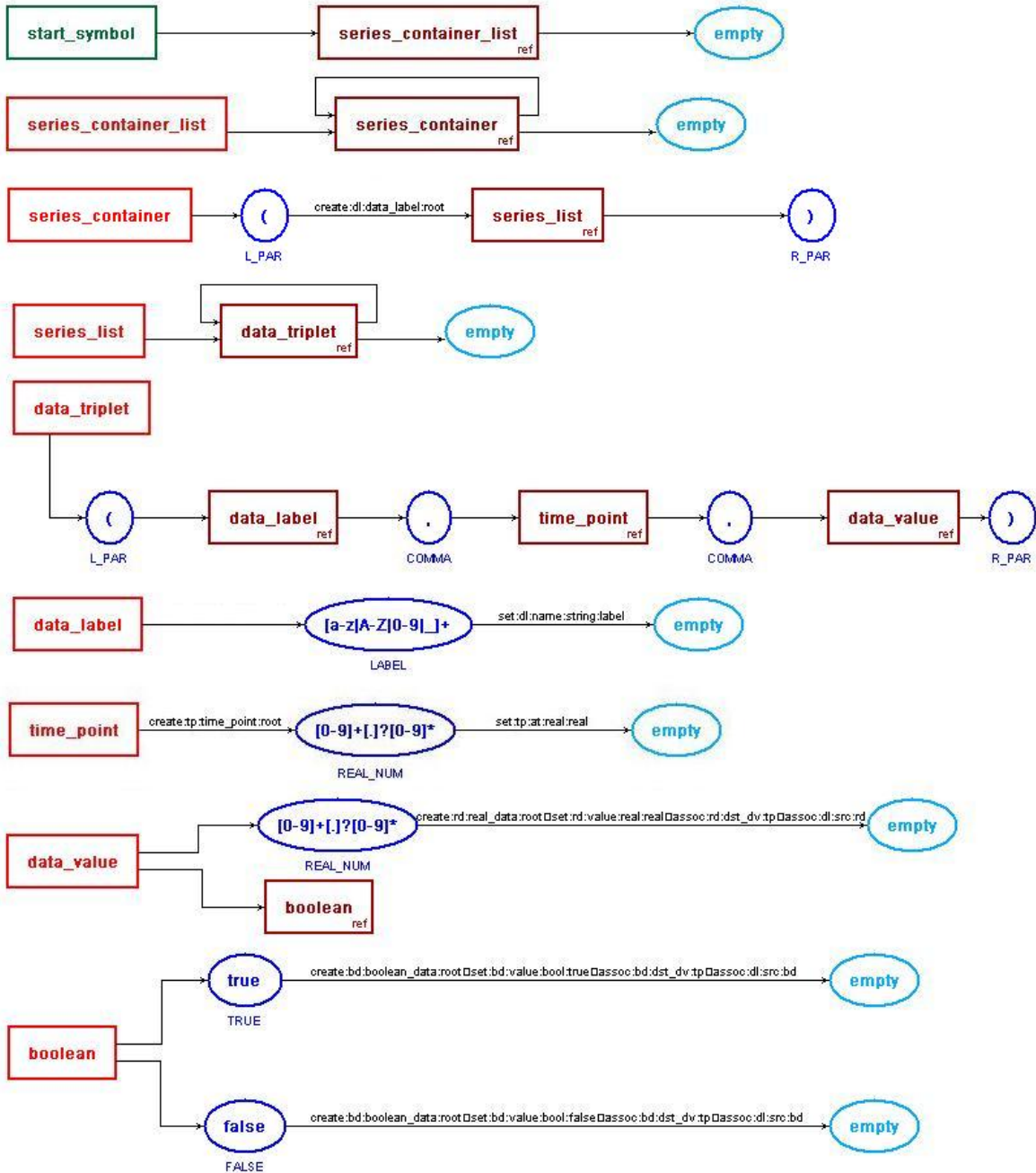
E. The output UDM state chart object network

```
<RootFolder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="StateChart.xsd">
  <OrState name="NewOrState" Marked="false" DefaultTransition="">
    <AndState name="AndState" Marked="false" DefaultTransition="">
      <OrState name="2_0" Marked="false" DefaultTransition="">
        <Init name="Init" Marked="false" DefaultTransition=""/>
        <State _id="id6" name="L0" Marked="false"
dstTransition="id8" srcTransition="id9" DefaultTransition=""/>
        <State _id="id7" name="L1" Marked="false"
dstTransition="id9" srcTransition="id8" DefaultTransition=""/>
        <Transition _id="id8" name="tr2" Guard="" Action=""
isSync="false" Trigger="" dstTransition="id7" srcTransition="id6"/>
        <Transition _id="id9" name="tr3" Guard="" Action=""
isSync="false" Trigger="" dstTransition="id6" srcTransition="id7"/>
      </OrState>
      <OrState name="2_1" Marked="false" DefaultTransition="">
        <Init name="Init" Marked="false" DefaultTransition=""/>
        <State _id="idc" name="M0" Marked="false"
dstTransition="ide" srcTransition="idf" DefaultTransition=""/>
        <State _id="idd" name="M1" Marked="false"
dstTransition="idf" srcTransition="ide" DefaultTransition=""/>
        <Transition _id="ide" name="tr4" Guard="" Action=""
isSync="false" Trigger="" dstTransition="idd" srcTransition="idc"/>
        <Transition _id="idf" name="tr5" Guard="" Action=""
isSync="false" Trigger="" dstTransition="idc" srcTransition="idd"/>
      </OrState>
      <OrState name="2_2" Marked="false" DefaultTransition="">
        <Init name="Init" Marked="false" DefaultTransition=""/>
        <State _id="idl2" name="K0" Marked="false"
dstTransition="idl4" srcTransition="idl5" DefaultTransition=""/>
        <State _id="idl3" name="K1" Marked="false"
dstTransition="idl5" srcTransition="idl4" DefaultTransition=""/>
        <Transition _id="idl4" name="tr7" Guard="" Action=""
isSync="false" Trigger="" dstTransition="idl3" srcTransition="idl2"/>
        <Transition _id="idl5" name="tr8" Guard="" Action=""
isSync="false" Trigger="" dstTransition="idl2" srcTransition="idl3"/>
      </OrState>
    </AndState>
  </OrState>
</RootFolder>
```

F. Grammar description of simple time series



G. Grammar description of time series with data triplets



H. The simple time series text

```
(
Cdc25_activation_Ca
(1.25, 0.8)
(2.5, 0.9)
(5.0, 1.0)
(10, 1.0)
)
(
Cdc25_inactivation_Ca
(5.0, 0.75)
(10, 0.5)
(20, 0.1)
(40, 0.0)
)
```

I. The output UDM simple time series object network

```
<Container xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="TimeSeries.xsd">
  <data_label _id="id2" src="id4 id6 id8 ida" name="Cdc25_activation_Ca"/>
  <data_label _id="idb" src="idd idf id11 id13"
name="Cdc25_inactivation_Ca"/>
  <real_data _id="id4" dst="id2" value="0.800000" dst_dv="id3"/>
  <real_data _id="id6" dst="id2" value="0.900000" dst_dv="id5"/>
  <real_data _id="id8" dst="id2" value="1.000000" dst_dv="id7"/>
  <real_data _id="ida" dst="id2" value="1.000000" dst_dv="id9"/>
  <real_data _id="idd" dst="idb" value="0.750000" dst_dv="idc"/>
  <real_data _id="idf" dst="idb" value="0.500000" dst_dv="ide"/>
  <real_data _id="id11" dst="idb" value="0.100000" dst_dv="id10"/>
  <real_data _id="id13" dst="idb" value="0.000000" dst_dv="id12"/>
  <time_point at="1.250000" _id="id3" src_tp="id4"/>
  <time_point at="2.500000" _id="id5" src_tp="id6"/>
  <time_point at="5.000000" _id="id7" src_tp="id8"/>
  <time_point at="10.000000" _id="id9" src_tp="ida"/>
  <time_point at="5.000000" _id="idc" src_tp="idd"/>
  <time_point at="10.000000" _id="ide" src_tp="idf"/>
  <time_point at="20.000000" _id="id10" src_tp="id11"/>
  <time_point at="40.000000" _id="id12" src_tp="id13"/>
</Container>
```

J. The time series text with triplets

```
(
(Cdc25_activation_Ca, 1.25, 0.8)
(Cdc25_activation_Ca, 2.5, 0.9)
(Cdc25_activation_Ca, 5.0, 1.0)
(Cdc25_activation_Ca, 10, 1.0)
)
(
(Cdc25_inactivation_Ca, 5.0, true)
(Cdc25_inactivation_Ca, 10, 0.5)
(Cdc25_inactivation_Ca, 20, 0.1)
(Cdc25_inactivation_Ca, 40, 0.0)
)
```

K. The output UDM time series with triplets object network

```
<Container xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="TimeSeries.xsd">
  <boolean_data _id="idd" dst="idb" value="true" dst_dv="idc"/>
  <data_label _id="id2" src="id4 id6 id8 ida" name="Cdc25_activation_Ca"/>
  <data_label _id="idb" src="idd idf id11 id13"
name="Cdc25_inactivation_Ca"/>
  <real_data _id="id4" dst="id2" value="0.800000" dst_dv="id3"/>
  <real_data _id="id6" dst="id2" value="0.900000" dst_dv="id5"/>
  <real_data _id="id8" dst="id2" value="1.000000" dst_dv="id7"/>
  <real_data _id="ida" dst="id2" value="1.000000" dst_dv="id9"/>
  <real_data _id="idf" dst="idb" value="0.500000" dst_dv="ide"/>
  <real_data _id="id11" dst="idb" value="0.100000" dst_dv="id10"/>
  <real_data _id="id13" dst="idb" value="0.000000" dst_dv="id12"/>
  <time_point at="1.250000" _id="id3" src_tp="id4"/>
  <time_point at="2.500000" _id="id5" src_tp="id6"/>
  <time_point at="5.000000" _id="id7" src_tp="id8"/>
  <time_point at="10.000000" _id="id9" src_tp="ida"/>
  <time_point at="5.000000" _id="idc" src_tp="idd"/>
  <time_point at="10.000000" _id="ide" src_tp="idf"/>
  <time_point at="20.000000" _id="id10" src_tp="id11"/>
  <time_point at="40.000000" _id="id12" src_tp="id13"/>
</Container>
```

REFERENCES

- [1] J. Sztipanovits, and G. Karsai, "Model-Integrated Computing", IEEE Computer, Apr. 1997, pp. 110-112.
- [2] Institute for Software Integrated Systems, <http://www.isis.vanderbilt.edu/>
- [3] GME 3 User's Manual, Institute for Software-Integrated Systems, Vanderbilt University, March 2003.
- [4] Arpad Bakay, Endre Magyari, "The UDM Framework", Institute for Software-Integrated Systems, Vanderbilt University, April 2003.
- [5] Agrawal A., Karsai G., Shi F.: A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations, ISIS-03-403, November, 2003.
- [6] Anantha Narayanan, Declarative Techniques for Unparsing Complex Data Structures, Thesis, ISIS, 2004
- [7] OMG Unified Modeling Language Specification, Version 2.0, October 2004.
- [8] The Object Management Group, <http://www.omg.org>
- [9] IBM Rational Software, <http://www-306.ibm.com/software/rational>
- [10] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.
- [11] Basic Class Diagram Elements (in UML)
<http://aliweb.cern.ch/offline/geant4/uml/UMLElements.html>
- [12] BHLex: A Programmable Lexical Analyser,
<http://www.codeproject.com/cpp/BHLex.asp>
- [13] Introduction to Grammars and Language Analysis,
<http://www.cs.binghamton.edu/~zdu/parsdemo/gramintro.html>
- [14] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley Pub Co, January 1986.
- [15] Noam Chomsky, "Three models for the description of language", IRE Transactions on Information Theory, 2 (1956), pages 113-124
- [16] Terrance John Parr, "ANTLR Reference Manual", January 20003,
<http://www.antlr.org/doc/index.html>

- [17] ANTLR plug-in for Eclipse, <http://antlrclipse.sourceforge.net/>
- [18] The Eclipse Platform, <http://www.eclipse.org/>
- [19] Voice Web Solutions, Visual Grammar Builder Studio,
<http://www.voicewebsolutions.net/products/gram/features.htm#4>
- [20] The ‘Logger’ Library , <http://www.nick.rozanski.com/logger.htm>
- [21] The GNU General Library License, <http://www.gnu.org/>
- [22] Time Series, <http://jigcell.biol.vt.edu/FEbio.html>
- [23] BioSpice, <https://users.biospice.org/home.php>