

To my source of Inspiration, My parents

&

To Pramila, the love of my life

ACKNOWLEDGEMENTS

The DARPA/IXO MOBIES program, Air Force Research Laboratory under agreement number F30602-00-1-0580 and NSF ITR on "Foundations of Hybrid and Embedded Software Systems" programs have supported, in part, the research described in this dissertation. Some tools described in this dissertation have been developed by other members of the MoBIES team. Feng Shi developed in part the Graph Rewriting Engine (GRE), Zsolt Kalmar developed the Graph Rewriting Debugger (GRD) and Attila Vizhanyo developed the Code Generator (CG).

To begin with I would like to thank Dr. Gabor Karsai my academic advisor and the chair of my dissertation committee. He has motivated and guided me through this endeavor. His knowledge and patience are virtues I can only dream of achieving. I am grateful to members of my dissertation committee, Dr. Janos Sztipanovits, Dr. Douglas Schmidt, Dr. Gautam Biswas, Dr. Jeremy Spinrad and Dr. Mark Ellingham for keeping my focus on the goals and for directing me back on track when I veered. The MoBIES team consisting of Dr. Gyula Simon, Dr. Sandeep Neema, Feng Shi, Attila Vizhanyo, Zsolt Kalmar, Andras Lang, Tamas Paka and Anantha Narayanan deserve my heartiest thanks for being the greatest teams to work with.

Last but definitely not the least I would like to thank mom and dad for believing in me all through the journey and encouraging me to push forward whenever I was tired. Without the training I have received from them I would never have reached where I am. The rest of my family, Jiten, Shilpa, Roma, Sandhir, Mona, Rashi and Sparsh has played a vital role in my endeavor. They have been by my side at every crossroad of life helping

me take the right decision. Finally I would like to thank my soul mate Pramila for without her the journey would definitely have been more challenging and the goals more difficult to achieve.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS.....	iii
TABLE OF CONTENTS.....	v
LIST OF TABLES.....	viii
LIST OF FIGURES.....	ix
LIST OF ABBREVIATIONS.....	xii
Chapter	
I. INTRODUCTION.....	1
II. BACKGROUND.....	6
Model Based Software Engineering.....	6
Model Classification.....	7
Low-Level and High-Level Modeling Languages.....	8
High-level models.....	16
Low-Level Vs High-Level.....	18
Domain-Specific and Domain-Independent Languages.....	19
Domain Specific Vs Domain Independent.....	24
Textual and Graphical Languages.....	25
Generative and Model Based Solutions.....	25
Generative Programming (GP).....	26
Model Integrated Computing (MIC).....	29
Summary of Model-Based Solutions.....	38
Graph Grammars And Transformations.....	39
Node Replacement Graph Grammars.....	40
Node Label Controlled (NLC).....	40
Neighborhood Controlled Embedding (NCE).....	42
Hyperedge Replacement Graph Grammars.....	46
Algebraic Approach to Graph Transformation.....	48
Programmed Graph Rewriting Systems.....	51
Programmed Structure Replacement Systems.....	55
Summary of Graph Grammars and Transformations.....	57
Graph Transformation Based Tools.....	58
PROGRES.....	58
AGG.....	59
Comparison of Features.....	60

Critique of Graph Transformation Tools	60
III. RESEARCH PROBLEM, HYPOTHESIS AND METHODS	63
Research Hypothesis.....	67
Research Methods.....	68
Completion Criteria	69
IV. GREAT: A MODEL-TO-MODEL TRANSFORMATION LANGUAGE	71
Heterogeneous Graph Transformations.....	71
Definitions	75
The Pattern Specification Language	76
Simple Patterns	76
Fixed Cardinality Patterns.....	78
Extending the Set Semantics.....	82
Cardinality for Edges	84
Variable Cardinality.....	84
Pattern Graph and Match Definition.....	86
Graph Rewriting/Transformation Language.....	87
Language Realization.....	89
The Language For Controlled Graph Rewriting And Transformation	90
Sequencing of Rules	93
Hierarchical Rules.....	94
Branching using test case.....	98
Non-deterministic Execution	100
Termination.....	102
Enabling Optimized Graph Transformations.....	102
Typed Patterns	102
Pivoted Pattern Matching.....	103
Reusing Previously Matched Objects	104
User Controlled Traversal.....	105
V. THE EXECUTION FRAMEWORK FOR GREAT.....	107
Concrete Syntax.....	108
Abstract Syntax.....	110
Execution Engine.....	112
Graph Rewriting Debugger (GRD).....	118
Code Generator	119
Comparison of CG with GRE.....	120
Integrated Development Environment.....	123
Model Development Tools	125
Execution Invocation Tools	126
VI. A CASE STUDY – SIMULINK/STATEFLOW TO HSIF	127
The Inputs and Outputs of the Semantic Translator	128

The output: HSIF	128
The input: A subset of the MSS language	128
Example: Tank Level Control.....	129
Implementing the Algorithm in GReAT.....	132
Translating Stateflow	132
Translating Simulink.....	138
Translating the Tank Level Control example	139
Summary	140
Conclusion	142
VII. RESULTS, CONCLUSIONS AND FUTURE WORK.....	143
Results.....	143
Requirement 1	143
Requirement 2	143
Requirement 3	143
Requirement 4	144
Requirement 5	144
Requirement 6	148
Requirement 7	149
Requirement 8	149
Requirement 9	149
Revisiting the Research Hypothesis and Completion Criteria.....	156
Conclusion	159
Future Work.....	163
Appendix	
A. ALGORITHM FOR SINGLE CARDINALITY PATTERN MATCHING	165
B. ALGORITHM FOR FIXED CARDINALITY PATTERN MATCHING	166
C. ALGORITHM FOR VARIABLE CARDINALITY PATTERN MATCHING ...	167
D. FORMAL SEMANTICS OF GREAT	171
E. CONFIGURATION ASPECT OF UMT	181
F. THE SIMULINK/STATEFLOW TO HSIF TRANSLATION ALGORITHM....	183
REFERENCES	187

LIST OF TABLES

Table	Page
1. Comparison of Various MIC Tools.....	37
2. Comparison of Graph Transformation Tools	60
3. Concrete Syntax of the pattern graph and the rule interface	109
4. Mapping Simulink blocks to sub expressions.....	139
5. Compilation of different projects developed in GReAT.....	158

LIST OF FIGURES

Figure	Page
1. An Example Petri Net	14
2. Basic notations of UML class diagrams [36]	18
3. an Example IDEF3 Process Description Diagram [30]	21
4. Design Methodology Management using Ptolemy [31]	23
5. The MIC Development Cycle [1].....	30
6. A NLC production.....	41
7. Application sequence of a production.....	42
8. A NCE production.....	43
9. Two graphs with embedding and the result of their substitution [54]	45
10. Hyperedge production	46
11. An example to demonstrate the hyperedge production.....	46
12. DPO production.....	50
13. SPO production and example	51
14. A production in the PROGRES system.....	55
15. Metamodel of hierarchical concurrent state machine using UML class diagrams. .	72
16. Metamodel of a simple finite state machine.....	73
17. A metamodel that introduces cross-links	74
18. Non-determinism in matching a simple pattern	77
19. Pattern specification with cardinality	79
20. Pattern with different semantic meanings	81
21. Conflicting match for the tree semantics.....	82
22. Hierarchical patterns using set semantics.....	83

23.	Pattern with cardinality on edge.....	84
24.	Variable cardinality pattern and family of graphs.....	85
25.	An example rule with patterns, guards and attribute mapping.....	90
26.	UML class diagram for the abstract syntax classes of GReAT: The core transformation classes	91
27.	UML class diagram for the abstract syntax classes of GReAT: The interface	92
28.	Firing of a sequence of 2 rules	94
29.	Rule execution of a Block	95
30.	Sequence of execution within a <i>Block</i>	96
31.	Rule execution sequence of a <i>ForBlock</i>	97
32.	Execution of a <i>Test/Case</i> construct	98
33.	Execution of a single <i>Case</i>	99
34.	Inside the execution of a <i>Test</i>	100
35.	A non-deterministic execution sequence.....	101
36.	Pivoted Matching	104
37.	Transformation Rule with pivot	104
38.	Sequence of rules with passing of previous results.....	106
39.	Concrete syntax of the different expressions in GReAT.....	108
40.	GR: the abstract syntax of GReAT	111
41.	High-level block diagram of GRE	114
42.	Block execution algorithm	115
43.	For block execution algorithm	116
44.	Test execution algorithm.....	117
45.	Algorithm for rule execution.....	118
46.	Performance graphs for $Df \rightarrow Fdf$	121
47.	Performance graphs for $Hsm \rightarrow Fsm$	122

48.	Block diagram of the GReAT IDE.....	124
49.	A tank with three valves.....	130
50.	The "true" (hybrid automata) state machine for the tank example.....	131
51.	The StateflowPart Rule.....	133
52.	The HSM2FSM rule.....	133
53.	Inside the OR rule.....	134
54.	<i>ElevateChildOr</i> rule.....	135
55.	The StateSplitting rule.....	136
56.	The SetImplicitValues Rule.....	137
57.	Stages of Stateflow splitting.....	140
58.	The domain of Turing machines.....	145
59.	The top-level rule of Turing machine.....	146
60.	Internals of <i>RunMachine</i>	146
61.	Inside Q1 block, choosing action for current state and symbol.....	147
62.	Action taken for a particular State, symbol pair.....	148
63.	Transformation to make isomorphic copy of graph.....	150
64.	Metamodel of the configuration aspect of UMT.....	181

LIST OF ABBREVIATIONS

UML – Unified Modeling Language

MDA – Model Driven Architecture

OCL – Object Constraint Language

DSL – Domain Specific Language

GPL – General Purpose Language

GP – Generative Programming

MIC – Model Integrated Computing

GReAT – Graph Rewriting And Transformation

GRE – Graph Rewriting Engine

GRD – Graph Rewriting Debugger

CG – Code Generator

MoC – Model of Computation

FSM - Finite State Machine.

CFSM - Codesign Finite State Machine.

TM – Turing Machine

HPN - Hierarchical Petri Nets.

SDF - Synchronous Data Flow.

ADF - Asynchronous Data Flow.

OMG – Object Management Group

AHEAD – Algebraic Hierarchical Equations for Application Design

DSDE – Domain Specific Design Environment

DOME – Domain Modeling Environment

GTDL – Graph Type Definition Language

ER – Entity Relation

KMF – Kent Modeling Framework

GME – Generic Modeling Environment.

NLC – Node Label Controlled

NCE – Neighborhood Controlled Embedding

DPO – Double Pushout

SPO – Single Pushout

PROGRES – PROgrammed GRaph REplacement System

PSRS – Programmed Structure Replacement System

DSMDA – Domain-Specific Model Driven Architecture

DSPIM – Domain-Specific Platform Independent Model

DSPSP – Domain-Specific Platform Specific Model

DSME – Domain-Specific Modeling Environment

HCSM – Hierarchical Concurrent State Machine

GSS – Grouped Set Semantics

UMT – UML Model Transformer

GR – Graph Rewriting

UDM – Universal Data Model

MSS – Matlabs Simulink and Stateflow

HSIF – Hybrid System Interchange Format

HA – Hybrid Automata

XML – eXtensible Markup Language.

XSD – XML Schema Definition

CHAPTER I

INTRODUCTION

The evolution of programming languages shows a clear direction towards higher levels of abstraction. This evolution started from assembly languages, went on to procedural languages, then to object-oriented languages and now the state of the art is component-oriented languages and frameworks. In the same timeframe, top down approaches classified as Model Based Software Engineering [7] tried to develop high-level graphical languages and generate assembly/machine code from them. These approaches attempted to bridge large semantic gaps between very-high-level semantic models and very-low-level languages. There were many challenges in such an enterprise and tool infrastructures and frameworks did not live up to expectations. This led to their failure to achieve the goals set by the community. This community found success in more rigorous domain-specific fields such as embedded systems where formal and graphical models were already in use. An example of such a success is Matlab's Simulink/Stateflow [38] modeling language. With the advent of Unified Modeling Language (UML) and Model Driven Architecture (MDA) that advocate the use of models in software development, the communities were brought together and are producing promising results.

Languages can also be divided into textual and graphical categories. Graphical languages are usually impractical for general-purpose programming but can be useful in a limited context in specific domains. We believe that a mixed textual and graphical notation can be helpful in limited domains. For example, in the software development

domain, the UML [3] specification has both textual (Object Constraint Language) and graphical (Use-Case Diagram, Class Diagram, etc.) notations. In hardware development domain, tool vendors [39] are now providing a graphical notation for the structural description of hardware while the behavioral description is still textual.

The primary reasons behind the limited success of Domain Specific Languages (DSLs) have historically been the following:

- DSLs are more expensive to create as the development cost and time is borne by a small user community
- Since there is a small user base, tools and support for a DSL is not at par with General Purpose Languages (GPLs) and
- The wide user base and longer life of GPLs helps make the language implementations robust and reliable.

For DSLs to become more popular, the three hurdles mentioned above must be addressed. A key limitation is the *cost of development* (in terms of time and effort), which we conjecture can be reduced by creating a framework for developing DSLs. This approach has several advantages. First, the framework can be used to develop many languages, and thus the cost and time of development is reduced and can be absorbed by a larger community. Second, the framework can be the focal point for a wider user base, thus making it profitable for industries to provide support and tools. Within the framework there will be a development cost for a given DSL that needs to be minimized for the framework to be effective.

In the field of textual languages and compiler design there exists vast literature on textual grammars, parsers, parser generators and other formal methods to specify and

implement textual languages and compilers. This helps automate the different stages of language development and makes it easy to develop and deploy new languages. For instance, regular expressions can be used to specify language lexemes and a lexical analyzer generator such as Lex [5] can be used to automatically generate the lexical analyzer. Similarly, a context free grammar specification can be used to specify the language grammar and a parser generator such as Yacc [5] can be used to produce the parser. These tools focus on converting a textual notation into tokens and then into trees. In graphical languages the starting point is a graph of objects that represents the program. The biggest challenge is in converting this graph into another graph (abstract syntax tree) of a known programming language. In compilers for textual languages this stage is usually straightforward but not much automation is available. For graphical domain specific languages theory and tools need to be developed for this stage of the compilation/translation.

Currently, areas such as Generative Programming (GP) [42] and Model Integrated Computing (MIC) [1] have tried to explore different methods for the specification of domain-specific languages and their compilers. Formal definition and automated implementation of model-compilers is one field in this domain that has vast potential. Design and development of such a system should be based on a sound mathematical foundation.

Extensions of grammars for textual languages to graphs have been proposed for over 20 years and have emerged into a field called graph grammars and transformations. Such a foundation can be used for the formal specification and automated implementation of model compiler and model-to-model transformers. However, these results have not

been applied to the development of methodologies or tools that facilitate the development of modeling languages. Applying theoretical work of graph grammars towards the design of modeling languages seems to have much potential. Although these concepts cannot be applied directly, they can be used as a foundation for addressing the needs and requirements of model transformations.

This dissertation shows how graph grammar and transformation can be used as the formal foundation to develop a model-to-model transformation language that can be used to specify and automatically implement model transformer. Such a language would address the current deficiency of MIC frameworks.

This dissertation is organized as follows: Chapter I is a survey in the fields of model-based software engineering. Generative Programming and Model Integrated Computing techniques are explored to study their strengths and weaknesses as a domain-specific modeling framework. It is followed by a review of the theoretical work on graph grammars and transformations in Chapter II to investigate if it may be able to solve some problems identified in the meta-programmable tools. A survey of some of the notable graph grammar and transformation based tools and evaluation of the tools is presented.

Chapter III summarizes the findings in the background section and states the dissertation proposal along with the goals, completion criteria and metrics for measuring success.

Chapter IV describes Graph Rewriting and Transformation (GReAT) a model-to-model transformation language based on the theoretical work of graph grammars and transformations. The language developed is divided into four parts and each part is described along with their motivation, design decisions and tradeoffs. The last part of the

Chapter compiles all the language features that were developed to increase the efficiency of the execution of the transformations.

Chapter V describes the tool infrastructure developed for GReAT. First, the concrete and abstract syntax of GReAT are described. Then, an interpreter for the GReAT called Graph Rewriting Engine (GRE), a debugger called Graph Rewriting Debugger (GRD) and a compiler that produces C++ code called Code Generator (CG) are discussed. This is followed by the description GReAT's integrated development environment.

Chapter VI describes the use of GReAT to solve a challenge problem chosen for study. The Chapter states the challenge problem, its input and output, the algorithm for solving it and the implementation in GReAT. Finally the Chapter draws some conclusions about GReAT based on this case study.

Chapter VII presents results to evaluate whether GReAT and its implementation meet the requirements stated in Chapter III. Conclusions are drawn and future directions of this work are explored.

CHAPTER II

BACKGROUND

Model Based Software Engineering

Software engineering is a discipline where a variety of challenges have to be met frequently. These challenges are mainly due to three causes: (1) System complexity: inherent complexity of the problem domain, logic and software development. (2) Implementation technology complexity and (3) Organization of the development process. A great deal of research effort has gone into each of the above mentioned areas and specifically in the area of *System complexity*. To mitigate system complexity there has been a continual quest to raise the level of abstraction used for the specification of such systems. By raising the level of abstraction the developer do not have to think about the finer details that the new abstraction hides and can focus their energies to building bigger, better and more complex systems.

This trend is apparent in programming languages that started from machine languages and have evolved to the state of the art in object-oriented and component-based languages. This quest has also given rise to various model-based techniques that use abstractions of the problem domain to specify the solution [7].

In engineering models are abstractions of the real world and are used to precisely describe and analyze the working of some relevant portions of the physical system. Some examples of models are: scaled models of buildings and cars that function in the same manner as their real life counter parts.

The main benefit of models is that they help in abstracting away irrelevant details while highlighting the relevant. Mathematical models are often found in many disciplines such as control engineering where the functioning of a plant is described in terms of differential equations. Such abstract and formal models are often used for analysis through simulation and formal verification.

The software engineering discipline has only recently begun to adopt modeling. Modeling languages that are suitable for expressing various aspects of software have been introduced and are being used in the community. Some example languages are dataflow and its variants [26][27][28], state machine and state charts [18][19][20][21], entity-relationship diagrams [34] and more recently object-oriented modeling languages such as UML [35]. Several tools have been implemented that allow graphical specification of the structure and behavior of software a subset of the formalisms mentioned above. These tools try to automate the process of simulation, verification and synthesis of the end system. [7]

In this Chapter various modeling techniques are studied to find their strengths and weaknesses. An attempt will be made to generalize these results to classes of languages. The modeling languages studied will be classified based on various criteria. Results of all the languages that fall into one class can then be studied to find commonalities. These commonalities should provide insight into the general properties that pertain to languages belonging to the class.

Model Classification

A model is an abstract representation of a system. There are various notations for the specification of models and models are used for varied purposes. Modeling languages

can be classified based on various attributes. One such classification can be based on level of detail captured in the models and it can vary from very precise low-level modeling languages to abstract high-level languages. Another classification can be based on the scope of the languages; it can vary from general purpose languages to highly customized domain-specific language. Modeling languages can also be classified based on their notation into visual, textual or both.

The rest of this Chapter will explore a few of these classifications and try to draw conclusions on the merits of the various classes of modeling languages.

Low-Level and High-Level Modeling Languages

One criterion for classification is the level of abstraction/detail captured by the models. On one hand low-level models, called Models of Computation (MoC) that precisely describe how computation is done. On the other hand high-level models are used to specify design and intention. In this context low-level languages are defined as those that are tightly coupled to the semantics of the underlying machine or to a MoC. High-level languages on the other hand are those where a numerous abstractions have been built on top of the underlying machine or languages that are closer to a problem domain. The next sub section will describe a few low-level and high-level modeling languages and compare them.

The formal definition of Models of Computation (MoC) is:

“A formal, abstract definition of a computer. Using a model one can more easily analyze the intrinsic execution time or memory space of an algorithm while ignoring many implementation issues. There are many models of computation which differ in computing power (that is, some models can perform computations impossible for other models) and the cost of various operations.”

[8]

Model of Computation (MoC) is a platform independent abstraction of a computing device. MoCs have precise execution semantics and are not tied to an implementation. Generating an implementation from an MoC for a particular platform is usually a simple and straightforward process. Examples of widely used MoCs are Finite State Machine (FSM), Turing Machine, Discrete-Event and Petri Nets. This section will review these MoCs.

Finite State Machine (FSM)

Finite State Machines (FSMs) are a state based model of computation where the behavior of the system only depends on the current state, and the current input. FSMs are defined as:

“A model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function. There are many variants, for instance, machines having actions (outputs) associated with transitions (Mealy machine) or states (Moore machine), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (nondeterministic finite state machine), one or more states designated as accepting states (recognizer), etc.”

[9]

Formally, a FSM is defined as a 5-tuple, $FSM = (S, \Sigma, T, s, A)$ where:

$\Sigma = \{e_1, e_2, e_3, \dots, e_n\}$ is an alphabet set

$S = \{s_1, s_2, \dots, s_m\}$ is a set of states

$T : S \times \Sigma \rightarrow P(S)$ is a transition function.

s an element of S is the start state

A a proper subset of S is a set of accept states

The Finite State Machine (FSM) representation is useful in describing applications that are tightly coupled with their environment. It is also suited for control-

dominated and reactive applications. However, concurrency is not easily captured and results in the exponential growth in the number of states with linear increase in degree of concurrency. This problem is known as the “state space explosion” problem. In order to overcome these weaknesses of classical FSM, a number of extensions such as hierarchy and concurrency have been developed. A few such variants are discussed in brief [18].

SOLAR [19], a design representation for high-level control flow dominated systems is an extension of the FSM representation. Concurrency is achieved by capturing parallel components of the system as separate FSMs that communicate with each. The communication between FSMs is either with the help of ports that are wired together or with the help of communication channels that implement a protocol. Each component can either be a FSM or be composed of smaller FSMs. Thus the model allows hierarchical decomposition of the system.

Codesign Finite State Machine (CFSM) [20] is another model based on the FSM. It is intended to describe embedded applications with low algorithmic complexity. Both hardware and software can be depicted using this model of computation. It can be used to partition and implement applications. The basic communication primitive is an event and the behavior of the system is defined as a sequence of events. The events are broadcasted and have zero propagation time. This model of computation is used as an intermediate representation that high-level languages can map to [18][20].

Statecharts by Harel [21] is another extension of FSMs that provides three major facilities, namely hierarchy, concurrency and communication. Statecharts are high-level Finite State Machines having AND and OR states. The AND states primarily achieve concurrency while the OR states are for representing hierarchy. Communication is based

on events that are broadcasted instantaneously. This representation is well suited for large and complex reactive systems.

Finite State Machines (FSMs) are a simple yet powerful MoC and can be used to represent a wide range of systems from digital logic to communication protocols. FSMs have been widely studied and there are well established analysis methods and tools such as SMV [10]. However, large problems with concurrent behavior become very difficult to express due to the “state space explosion” problem. This prompted the introduction of a number of FSM variants. These variants have introduced a number of abstractions to deal with concurrency and mitigation of complexity. SOLAR, CFSM or Statecharts can be considered as high-level modeling formalisms useful for the specification of large problems. Functions can be specified that map these high-level representations to FSM which is a domain-independent, platform-independent MoC. By writing these transformations, users not only benefit from the use of high-level modeling languages but also from the verification capabilities of the low-level MoC. Furthermore, the abstract FSM representation can then be converted to a platform-specific implementation suitable for a particular platform. This helps in isolating the implementation from its intended behavior as specified by the requirements.

Turing Machine

A Turing machine is a computational device that is based on the notion of a tape, a read/write head and a controller for the head that is based on a finite state representation. It is defined as:

“A model of computation consisting of a finite state machine controller, a read-write head, and an unbounded sequential tape. Depending on the current state and symbol read on the tape, the machine can change its

state and move the head to the left or right. Unless otherwise specified, a Turing machine is deterministic.”

[11]

Formally, a Turing Machine (TM) is represented as a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where

$Q = \{q_0, q_1, \dots, q_m\}$ is a finite set of states

$\Sigma = \{s_1, s_2, \dots, s_n\}$ is a finite set of symbols called the input alphabet

Γ is a super set of Σ is a finite set of symbols called tape symbols

q_0 is an element of Q is the initial state

$\delta : Q \times (\Gamma \cup \{\Delta\}) \rightarrow (Q \cup \{h\}) \times (\Gamma \cup \{\Delta\}) \times \{R, L, S\}$ is a transition function

Here Δ denotes the blank and R, L and S denote move the head right, left and do not move it, respectively and h denotes the halt state.

A Turing machine consists of an infinite single dimensional tape, a read/write head and a finite state machine controlling the actions of the head. Each cell of the tape can contain a binary digit (0 or 1). Based on the current state and value at cell, an action is performed. The action can write a new value at the current location and possibly move the head by one position in either left or right direction. It can also change the state of the machine by taking a transition to another state.

This simple machine is a complete abstraction of a computing device. A Turing machine can solve any problem that can be expressed as a general recursive function [12] and Turing completeness is used as a measure of the expressiveness of programming languages. In practice, programming Turing machines is quite cumbersome. Instead high-level Turing complete programming languages such as C, C++ are used for programming needs.

Discrete-Event Systems

Systems that have discrete states and are driven by events over a period of time are referred to as Discrete-Event Systems [22].

Discrete Event Systems are described at an abstraction level where the time base is continuous (R), but during a bounded time-span, only a finite number of relevant events occur. These events can cause the state of the system to change. In between events, the state of the system does not change.

Hans Vangheluwe

These systems are asynchronous in nature and react to the discrete events over time. An event is considered instantaneous, that is the transition and actions are performed in zero time. As opposed to FSMs, Discrete event systems are not restricted to finite number of states. In Discrete event systems an event is tied to time while this may not necessarily be the case for FSMs.

Events over time are the primary method of communication between tasks. The events are time stamped and are sorted and processed in chronological order. Discrete-Event Systems are backed with formal mathematical descriptions [23] that facilitate formal verification and construction of deterministic systems. Though these systems are well suited for real-time applications, the primary disadvantage is the computational cost of sorting the events globally to maintain the chronology.

Petri Nets

Petri-Nets [24] is a graphical representation introduced by Carl Adam Petri. Petri Net is a mathematical tool that can be used to represent diverse semantic domains ranging from data-dominated to control-dominated applications. Semantics can be added to the models according to the domain. Some example domains are communication protocols,

distributed software, compilers and operating systems. A Petri-Net is described as a 5-tuple, $PN = \{P, T, F, W, Mo\}$ where:

- $P = \{p_1, p_2, p_3, \dots, p_m\}$ is a finite set of places
- $T = \{t_1, t_2, t_3, \dots, t_m\}$ is a finite set of transitions
- F is a subset of $(P \times T) \cup (T \times P)$ is a set of arcs giving flow relations
- $W: F \rightarrow \{1, 2, 3, \dots\}$ is the weight function
- $Mo: P \rightarrow \{0, 1, 2, \dots\}$ is the initial marking

Places hold tokens, and a transition occurs when the number of tokens required for the transition is present in the required places. A transition removes a specific number of tokens from its source and adds tokens to its destination. The number of tokens at each *place* in the Petri Net defines its state.

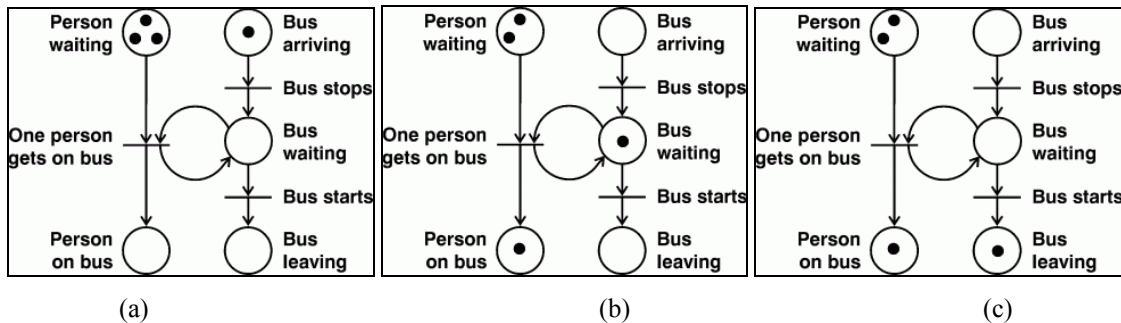


Figure 1. An Example Petri Net

Figure 1 shows three stages of an Example Petri Net. Figure 1(a) shows a state where there are three people standing on a bus stop and the bus is arriving; Figure 1(b) shows the net after two transitions have taken place. The first transition causes one token to move from the 'Bus arriving' place to the 'Bus waiting' place, the second transition

causes one token to move from ‘Person waiting’ to ‘Person on bus’ place. Finally Figure 1(c) shows the state of the net after the transition from ‘Bus waiting’ to ‘Bus leaving’.

The primary features of Petri Nets are concurrency and asynchronicity. Another advantage is that a number of mathematical analyses that can be performed on them.

However, the lack of hierarchy makes Petri Nets difficult to be used for developing large systems. Hierarchical Petri Nets (HPNs) [25] have been developed to mitigate the complexity of a flat representation. HPNs are modeled using bipartite directed graphs with inscription on the nodes and edges. There are two types of nodes, *transitional nodes* that represent activity and *places* that represent data or the state of the system. This approach extends the Petri Net semantics with hierarchy making it suitable for complex systems.

Data Flow Graph

The classical programming structure of computer-based systems is control based as described by the Von Neumann machine. An alternative approach is data-dominated where the control flow is determined by availability of data. These systems have nodes describing computation, and edges between nodes denoting a data path. If a node has sufficient data available on its incoming edges then it is ready to fire and will use the input data to generate output data. Transfer of data between computational modules is typically done with the help of buffers. This allows the tasks to run independently.

Formally, a dataflow diagram can be represented as a tuple (C, Df, s) where

$C = \{c_1, c_2, \dots, c_n\}$ is a set of nodes and

$Df : C \rightarrow C$ is a dataflow relation that captures data dependency

$s : Df \rightarrow \text{Integers}$ is the initial tokens function that defines the initial set of tokens in each dataflow relation.

There exist a number of variants of data flow. The two popular and distinct ones are Synchronous Data Flow (SDF) and Asynchronous Data Flow (ADF). In SDF the number of token produced and consumed by each node is fixed and needs to be known at the system design stage. This requirement allows SDF to be statically scheduled [28]. A static schedule is one that can be computed offline, has a finite sequence of execution of the nodes and requires bounded buffers where the maximum size of the buffers is known beforehand. ADF is defined as a data flow graph with unbounded buffers where computations can produce and consume variable number of tokens. Since the consumption and production of tokens can change at runtime, ADF cannot be scheduled statically and results in a greater run-time cost. However, it can be used to represent a large number of systems and is more flexible than SDF. Many extensions have been proposed to augment the data flow representation with hierarchy, strong data typing of tokens and parameterized nodes.

High-level models

High-level models describe systems at a higher level of abstraction. These models may be specified with few or no implementation details. A few examples are partial differential equations that specify the behavior of a controller [29], block diagrams that define the design of system artifacts, and high-level state machines that convey the basic behavior of a system. System level modeling languages fall in this category.

With recent advances in generation technologies, there is a push towards turning high-level languages such as UML into executable artifacts. An executable model is one

where sufficient information is captured to facilitate the synthesis of low-level details. This section will highlight UML as a widely used system-level modeling language.

Unified Modeling Language (UML)

Unified Modeling Language (UML) [35] is an Object Management Group (OMG) standard for diagrammatically representing object-oriented designs. UML consists of a number of diagrammatic representations and an UML class diagram is one of them. Class diagrams graphically represent classes along with their member variables and functions. Inheritance, aggregation and other associations are also graphically represented. These class diagrams are a standardized and clean way to represent the design of complex systems. The important aspects of diagrammatic representation are discussed here to provide a quick overview.

Figure 2 depicts the basic notations. A class is represented with the actual name of the class in place of the Class Name text. The name in angular braces depicts the stereotype of the class. A stereotype states that the class conforms to the strict rule defined by the stereotype. For example, in this paper the stereotype <<atom>> is used. The atom stereotype states that classes that confirm to the <<atom>> stereotype should not contain other classes. Attributes and operations are listed in separate containers within the class rectangle.

Class specialization is depicted using a triangular connector called ‘discriminator’. The class connected to the top of the triangle is the supertype while the classes connected to the bottom are subtypes. Associations between classes are depicted by a line between the classes. The roles the classes play in an association and their cardinality can be specified on the association. Alternatively an association class can be

specified to capture more information about the association. Composition represents a special kind of association. It specifies that a composer class contains instances of the composed class and these instances cannot exist outside the composer class. The composition is depicted with a line having a solid diamond towards the composer class. The role and cardinality of the composing class is also specified on the composition line. Cardinality specifies the numeric range of objects that are part of the association. For example, class A is composed of 2 instances of class B, then the cardinality of class B in this composition is said to be 2. Cardinality can be specified as a fixed number or a possible range of numbers. The different notations of cardinality are shown in Figure 2.

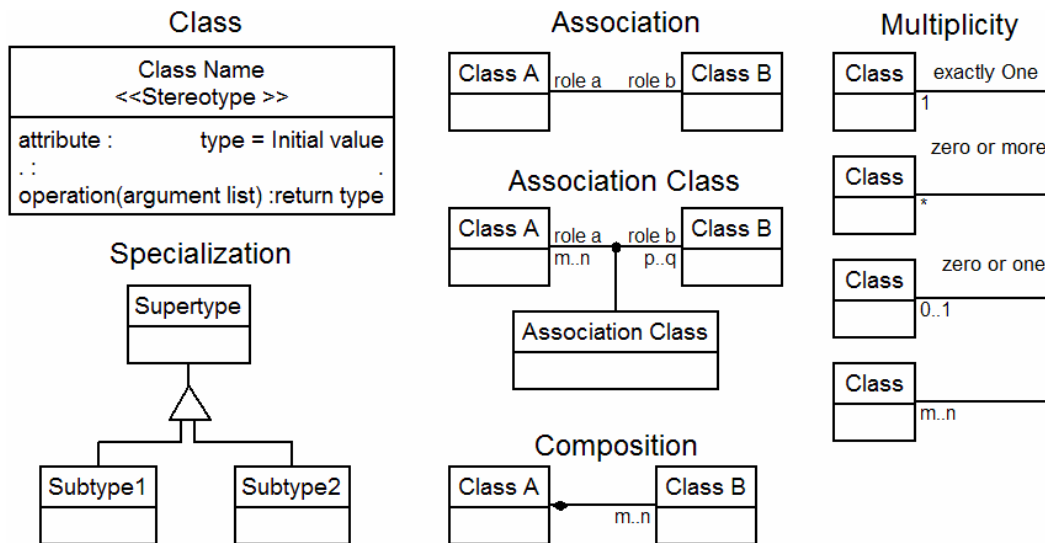


Figure 2 Basic notations of UML class diagrams [36]

Low-Level Vs High-Level

In the previous sections we have reviewed a few representative low-level and high-level modeling formalisms. Low-level MoCs are useful for precise specification of computing systems and have direct mappings to a computing device. For instance, FSMs

can be implemented using either digital logic or software. Algorithms for mapping a MoC to an implementation can be and have been in many cases, fully automated. Thus, we can view the low-level MoCs as executable computing devices. However, we have seen that these low-level MoCs are not an efficient method for the specification of large systems and sometimes do not scale well in their representation. For example, FSMs cannot be used for the specification of large parallel systems because of the state space explosion problem. Thus, other representations that help mitigate complexity such as State Chart and UML are used to describe large systems.

There exists a gap between the MoCs like FSM and Turing machine, and high-level representations such as architecture level block diagrams, State Charts and UML. In some cases humans are required to comprehend the problem and its solution using high-level representations and then encode the solution using programming languages such as C++ or Java. Some systems provide automated or semi automated programs to convert the high-level specification to the equivalent executable code. However, these translator programs are often difficult to develop and require a large amount of programming time, and effort.

Domain-Specific and Domain-Independent Languages

In many domains such as business or scientific computing, the high-level representations as well as the MoCs can be very specific. In such cases customized tool suites are required to leverage the benefits of the domain and to significantly increase productivity.

This leads to another classification of modeling languages based on the scope. Models can be classified as either domain-specific or domain independent. The definition

of a domain makes a great difference in this classification. In this paper the universal set is considered as the computing domain of computer programs i.e. general recursive functions. In this context a domain-specific computing paradigm will be one that does not span the entire computing space and/or one that makes a set of assumptions based on the domain.

Domain-Specific Modeling

Modeling formalisms discussed in the previous section were universal. They encompass large domains such software design or the domain of computability. Modeling formalisms that are tailored for a specific domain help users specify systems using domain concepts they are familiar with. Domain-specific modeling also allows users to specify systems at a higher level of abstraction. In such restricted domains, executable systems can be synthesized from high-level abstractions as domain knowledge can be used to fill in implementation details. There are many successful domain-specific languages available, for example Matlab Simulink/Stateflow, Ptolemy and IDEF3. This section describes a few of these domain-specific modeling formalisms.

IDEF3 - Process Flow And Object State Description Capture Method

The IDEF3 Process Description Capture Method provides a mechanism for collecting and documenting processes. IDEF3 captures precedence and causality relations between situations and events in a form natural to domain experts. This is achieved by providing a structured method for expressing knowledge about how a system, process, or organization works [30].

IDEF3 captures the behavioral aspects of an existing or proposed system. Captured process knowledge is structured within the context of a scenario, making

IDEF3 an intuitive knowledge acquisition device for describing any system. IDEF3 captures all temporal information, including precedence and causality relationships associated with enterprise processes. The resulting IDEF3 descriptions provide a structured knowledge base for constructing analytical and design models. These descriptions capture information about what a system actually does or will do and also provide for the organization and expression of different user views of the system [30].

There are two IDEF3 description modes, process flow and object state transition network. A process flow description captures knowledge of "how things work" in an organization, e.g., the description of what happens to a part as it flows through a sequence of manufacturing processes (see Figure 3). The object state transition network description summarizes the allowable transitions an object may undergo throughout a particular process. Both Process Flow Description and Object State Transition Description contain units of information that make up the system description. These model entities, as they are called, form the basic units of an IDEF3 description. The resulting diagrams and text comprise what is termed a "description" as opposed to the focus of what is produced by the other IDEF methods whose product is a "model." [30]

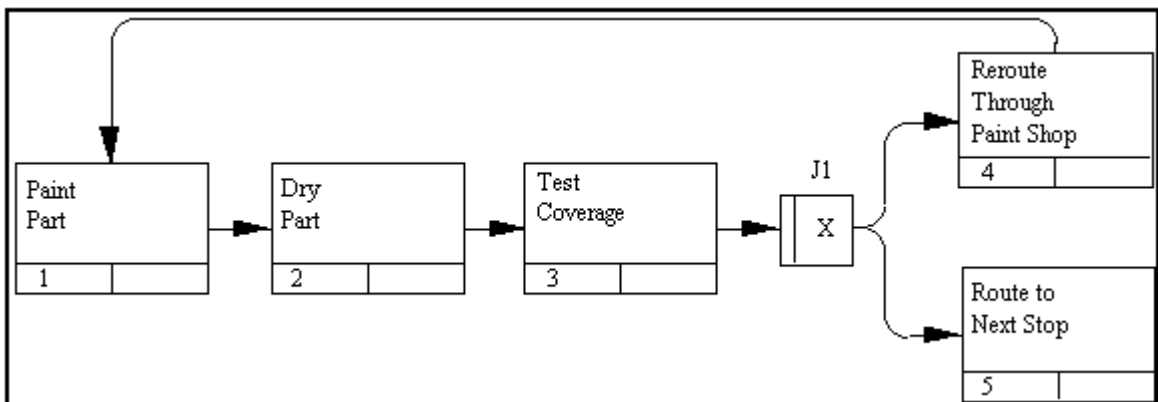


Figure 3 an Example IDEF3 Process Description Diagram [30]

An IDEF3 Process Flow Description captures a description of a process and the network of relations that exists between processes within the context of the overall scenario in which they occur. The intent of this description is to show how things work in a particular organization when viewed as being part of a particular problem solving or recurring situation. The development of an IDEF3 Process Flow Description consists of expressing facts collected from domain experts in terms of the basic descriptive building blocks. [30]

Ptolemy II – A Polymorphic Design Environment

Ptolemy is a project dedicated to the modeling, simulation and design of real-time, embedded applications that started in 1990 at University Of California at Berkeley. The focus of Ptolemy is on component-based design. The philosophy of this project is using different models of computation and developing an environment that allows the mixing of these models of computation to create a heterogeneous application [33].

Ptolemy is a polymorphous modeling tool used for the simulation of embedded applications. Figure 4 shows the design management strategy proposed by the Ptolemy project. Design starts with application specification using different models of computation and constraints. Different tasks of the system are evaluated and estimates are drawn. These estimates decide the hardware and software partition of the application. This is followed by hardware and software synthesis and verification. The final stage is the integration and system wide simulation [31].

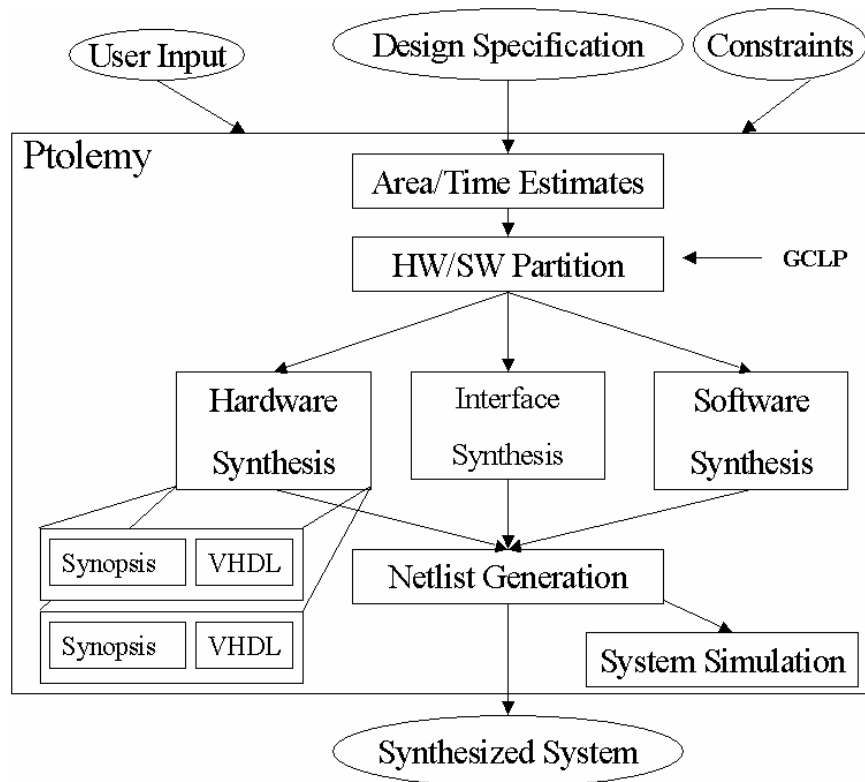


Figure 4 Design Methodology Management using Ptolemy [31]

A Java-based framework called Ptolemy II has been developed that implements the project ideas. The framework has an environment for the simulation and prototyping of heterogeneous systems. It is an object-oriented system allowing interaction between diverse models of computation. The Ptolemy software is extensible and publicly available. It allows experimentation with various models of computation, heterogeneous designs and co-simulation. The primary feature of Ptolemy is the facility to compose various models of computation. Some of the models of computation supported by Ptolemy are hierarchical finite state machine, data flow graphs, discrete-event and synchronous/reactive systems. After specifying the application using heterogeneous models, the next step is to partition the application. This is done using different partitioning algorithms like GCLP [32]. Ptolemy facilitates mixed mode system

simulation and synthesis. Software synthesis is supported for various models of computation along with support for composing these models. Hardware portions of the application are synthesized to VHDL. A register transfer level simulator (THOR) has also been added for simulating hardware applications [33].

Other key features of the project are the representation of modern theories in a block diagram specification, a modular approach, a mathematical framework for comparison of models of computations, and simulation and scheduling of complex heterogeneous systems [32].

Domain Specific Vs Domain Independent

Domain Specific Languages (DSLs) can increase productivity by bringing power programming to domain users via familiar specialized notations and languages [97]. It is well known that GPLs have been more prevalent and successful compared to DSLs, even though claims about DSLs' capabilities to increase productivity are widely accepted [37]. The primary reasons behind the limited success of DSLs have historically been the following:

- DSLs are more expensive to create as the development cost and time is borne by a small user community
- Since there is a small user base, tools and support for a DSL is not at par with GPLs
- The wide user base and longer life of GPLs helps make the language implementations robust and reliable.

Textual and Graphical Languages

Another view of languages characterizes them as textual and graphical. Graphical languages are usually impractical for general-purpose programming but can be useful in a limited context, in specific domains. One of the most successful recent example of a graphical domain-specific language is Matlab/Simulink [38] used for simulation and control engineering. The notation can use both textual and graphical parts according to the requirements of the target domain. For example, in the software development domain, the UML [35] specification has both textual (Object Constraint Language) and graphical (Use-Case Diagram, Class Diagram, etc.) notations. In hardware development domain, tool vendors [39] are now providing a graphical notation for the structural description of hardware even though the behavioral description is still textual.

For DSLs to become more popular the three hurdles mentioned above must be addressed. A key limitation is the cost of development (in terms of time and effort). The next Chapter is dedicated to various methods for automated generation of software.

Generative and Model Based Solutions

This section studies literature on various generative methods for software development as well as model based solutions.

Compiler compilers such as LEX [40] and YACC [41] are representative of the first breed of programs that were used for the automated implementation of programming languages. Since then a lot of progress has been made in both automated generation tools and the languages they develop. A few notable generative fields are Generative Programming and Model Integrated Computing (MIC).

Generative Programming (GP)

Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.

[42]

Generative programming focuses on the automation of assembly lines for software product families. They elevate the engineering discipline from development of single products to the development of product line for a family of products. The salient features of generative programming are (1) means to specify family members, (2) implementation components and (3) knowledge that maps the family member specification to the finished product [42].

The process starts with the analysis of a domain. Commonalities and variabilities within the domain are defined. Using the domain knowledge a common architecture of the domain is designed and a production plan is formulated. Finally the architecture, reusable components, domain-specific languages, generators and the production process are implemented [42].

Generative programming can be implemented in various ways ranging from code level methods such as generic programming, meta programming and aspect-oriented programming to high-level methods such as domain-specific languages/generators and intentional programming.

Generic programming is a term used when a program, module or component is built such that it is configurable. Configurability can be achieved by (1) the use of parameters and (2) programming using generic reusable abstract types and algorithms. Meta programming can be viewed as a special case of generic programming where the

programs are written with a lot of built-in variability such that the variability can be configured at different stages in the program lifecycle. Aspect-oriented programming is a method of abstracting out various aspects of a program. For example, the security issues of a program could be captured in a different aspect, making it easy to build systems with or without security.

Domain-specific languages are different than the code based approaches as they define a language based exclusively for the domain. The development of the infrastructure usually required considerable effort. Intentional programming is a new style of programming where the program is captured a set of intentions where intentions are the building blocks of the language. Intentions are extensible and users are free to write their own intentions.

Low-level methods can be used in the implementation of the platform and reusable components but the assembly and deployment of products still require high level methods.

The remainder of this section will discuss generator technologies such as Draco and GenVoca.

Draco

Draco is a methodology that encourages the development of domain-specific languages and tools for creating software. It was developed by James Neighbors at University of California, Irvine in 1980 [43][44].

In Draco the development cycle starts with a *Domain Analyst*, a person who has built many systems in a given domain. The *Domain Analyst* describes the variability of the domain by defining a *Domain Language* for expressing systems in the domain. The

next step is to define a visualization technique that makes the domain language readable to the user. This step is called *Prettyprinter Generation*. The next step is to specify optimizations in the form of *Source-to-Source Transformations* [43][44].

After the domain language and the associated tools are developed the next phase in the development process for the *Domain Designer* is to use the language and tools to create components and build libraries of components. Components are described as a set of refinements. The *System Analyst* then uses the language and libraries and extends the libraries to describe the required system. After the system has been described, a *System Specialist* uses the transformations in an interactive manner to convert the specification into executable code [43][44].

Genvoca

GenVoca is a tool and methodology developed by Don Batory. The tool is based upon step-wise refinement of domain-specific representation of the system. The next generation of the tool suite called AHEAD (Algebraic Hierarchical Equations for Application Design) [45] has been recently released.

The key theme is the composition of features to construct finished products. Features are the reusable building blocks of the product family. Various layers of abstraction of a product family are identified. A high-level layer then becomes a parameter of its lower-level layer. Then components are defined that form families of alternatives at each layer. Each product is a particular configuration within the family. The primary challenges are identifying the layers and implementing the various components. The layered approach helps build a progressive infrastructure from very generic configurable components to highly customized domain-specific systems.

Summary

Generative programming techniques are well-suited for the automation of well defined and tightly integrated product families. If the product family's specifications change drastically, then a lot of rework is required. Furthermore, support for design and analysis of new systems or families of systems is lacking.

Model Integrated Computing (MIC)

MIC is a software and system development approach that advocates the use of domain-specific models to represent relevant aspects of a system. The models capture system design and are used to synthesize executable systems, perform analysis or "drive" simulations. The advantage of this methodology is that it expedites the design process, supports evolution, eases system maintenance and reduces costs [1].

The MIC development cycle (see Figure 5) starts with the formal specification of a new application domain. The specification proceeds by identifying the domain concepts, together with their attributes and inter-relationships through a process called metamodeling [1]. Metamodeling is enacted through the creation of metamodels that define the abstract syntax, static semantics and visualization rules of the domain. The visualization rules determine how domain models are to be visualized and manipulated in a visual modeling environment. Once the domain has been defined, the specification; i.e. the metamodel of the domain is used to generate a Domain-Specific Design Environment (DSDE) through the step called "Meta-Level Translation". The DSDE can then be used to create domain-specific designs/models; for example, a particular state machine is a domain-specific design that conforms to the rules specified in the metamodel of the state machine domain. The next step is to do something useful with the models such as to

synthesize executable code, perform analysis or drive simulators. This is achieved by converting the models into another format such as executable code, input language of analysis tools, or configuration files for simulators. This mapping of the models to another useful form is called model transformation and is performed by model transformers [1]. Model transformers (also called “model interpreters”) are programs that convert models in a given domain into models of another domain. For instance, a source model can be in the form of a synchronous dataflow network of signal processing operations, while the target model can be in the form of Petri-nets, suitable for predicting the performance of the network. Note that the result of the transformation can be considered as a model that conforms to a different metamodel: the metamodel of the target [1].

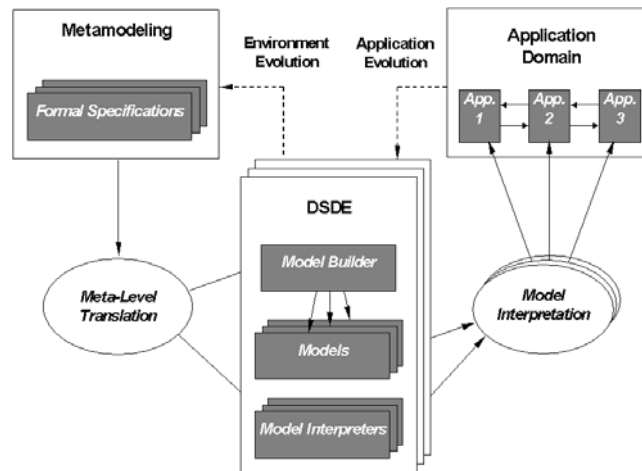


Figure 5 The MIC Development Cycle [1]

A Model Integrated Computing (MIC) implementation must have the following features.

1. **Meta framework tools** that will be used to describe the syntax, semantics and visualization of DSLs. The meta framework must provide support for the specification of a language defined by its abstract syntax, concrete syntax, static semantics, dynamic semantics, and visualization. The syntax of a programming language describes the structure of programs without considering their meaning. The abstract syntax of the language captures the abstract concepts used in the language and their relationships. Issues such as type-compatibility are captured in the static semantics of the language. Dynamic semantics are defined as the relation of the abstract syntax to a model of computation. In other words, it can be considered as a mapping from one language to another (provided the model of computation is captured in a linguistic framework).
2. **Language framework tools** that will be used for the creation, visualization and verification of sentences in a domain-specific language. The language framework should allow the use of the language in an integrated development environment that includes editing and visualizing instances of the language. The framework needs to enforce the concrete syntax and static semantics of the language during instance creation to provide maximum productivity. The final requirement of the framework is to be able to use transformation tools that map sentences of the language into sentences of some model of computation [8]. Examples of such models of computation are stack machines, process networks, finite state machines, etc. Often, although not always, sentences expressed in the target model of computation are executable, hence they are called “executable models”.

MIC promotes a metamodel-based approach to system construction, which has gained acceptance in recent years. The flagship research products following this approach are: GME [14], Atom3 [15], DOME [16], Moses [17]. Each implementation has a metamodeling layer that allows the specification of a domain-specific modeling languages and a modeling layer that allows the creation and modification of domain models.

The Domain Modeling Environment (DOME)

The Domain Modeling Environment (DOME) is a research project at Honeywell Technology Center. DOME has a metamodeling language called “DOME Tool Specification”. This is a proprietary language similar to entity relationship diagrams. There are two main entities the user can specify, a node and a link. The node represents a labeled node in the target language while a link represents a labeled directed edge in the target language. Links can be associated with nodes representing a constraint restricting the edge to be incident upon a particular kind of node. The language has inheritance; nodes can inherit properties from other nodes. Link associations and compositions are all described as attributes of nodes or other entities and the visualization does not show these associations. Nodes and links can have attributes called “Properties”, these properties can be typed.

Visualization specification is based on a set of basic shapes provided by the environment. The set of basic shapes consists of geometric shapes like square, circle, rectangle and a few others. The color of node and link types cannot be preset or chosen at metamodeling time.

There is no support for the specification of static semantics. Thus rules based on the value of attribute or based on particular pattern of objects cannot be specified in the metamodeling language. However, the user can write functions to implement such functionality that is triggered by GUI events.

There is no support for the specification of dynamic semantics. Dynamic semantics is represented and implemented by means of code written in a programming language called *Alter*.

Moses

Moses is a modeling, simulation, implementation and verification framework funded by Swiss Federal Commission for Technology and Innovation (KTI) and developed by the collaboration between Computer Engineering and Networks Lab, ETH Zurich, Switzerland and ESEC S.A., Cham, Switzerland [46].

Moses has a textual metamodeling language called Graph Type Definition Language (GTDL) [48]. The language is used to specify formalisms (modeling language). GTDL allows the specification of abstract and concrete syntax of the formalism. Vertex and Edge types can be defined in the language. Attributes can be defined as a type-name pair. These attributes can then be associated with vertex and edge types. Composition is represented by the parent graph type containing an attribute that of the child type [48].

Visualization information of the object is also declared, such as shape, border color, fill color, and dimensions [47]. Moses support for static semantic constraints or the lack thereof is not clear from either the documentation or direct experimentation with the tool [47]. Dynamic semantics are expressed in the form of Java code. They have a

simulation platform called Hades that can be extended to support specialized computation for each modeling language [47]. Moses supports animation of models by providing an extensible base animator class as part of the framework [47].

Atom³

Atom³ is a multi-paradigm modeling tool developed at Modeling, Simulation and Design Lab (MSDL) in the School of Computer Science at McGill University, Canada. Like other MIC implementations, it also supports metamodeling. The metamodeling language used by Atom³ is Entity Relationship (ER) diagrams. ER diagrams are used to specify the types of entities and their relations allowed in a particular modeling language. Typed attributes can be associated with each entity/relationship type. There is no direct support for composition or aggregation of entities or relations. The formalisms designed using this approach can be considered as flat representations [49][50].

Static semantics can be specified in the form of either OCL expressions or Python scripts which are associated with entities or relations. The user can also specify constraints to be pre or post conditions of an editing event [49][50].

Dynamic semantics are represented using a graph grammar based transformation specification. The specification is converted to a Python implementation [49][50].

Kent Modeling Framework

The Kent Modeling Framework (KMF) is under development at the Computing Laboratory, University of Kent at Canterbury, England. KMF uses UML 1.3 and XMI 1.0 as the metamodeling language. UML class diagrams and constraints are fed to ToolGen which in turn creates a set of Java files to implement the editor for the desired modeling language. The generated Java file can then be compiled to generate a modeling language

specific GUI. The lack of proper documentation hindered the successful generation of a modeling language [51].

MetaEdit+

MetaEdit+ is a professional tool developed by MetaCase, Finland, that allows the specification and implementation of modeling languages. The metamodeling language is based on a set of dialog boxes that allow the user to specify domain objects and their relations. Properties can be associated with objects. Only flat modeling languages can be built using these tools and hierarchy is not supported [52][53].

For visualization MetaEdit+ has visualization editor that allows the user to draw the visual representation of the objects and to as specify the visualization of properties [52][53].

Apart from defining the types of objects allowed in the modeling language MetaEdit+ supports the definition and use of libraries of object types. These can then be used by the domain modeler to assemble a custom language [52][53].

Generic Modeling Environment (GME)

The Generic Modeling Environment (GME) is the main component of the latest generation of MIC technologies developed at the Institute for Software Integrated Systems (ISIS), Vanderbilt University, USA. GME provides a framework for creating domain-specific modeling environments [1]. An important distinguishing property of the metamodeling environment of GME is that it is based on UML class diagrams [35] which is an industry standard. UML class diagrams are used to capture the syntax, semantics and visualization rules of the target domain. The meta-interpreter interprets the metamodels and generates a configuration file for GME. This configuration file acts as a

meta-program for the (generic) GME editing engine, so that GME behaves like the specialized modeling environment supporting the target domain. The core of GME is used both as the metamodeling environment and the target environment; the metamodeling language is just another domain-specific language that the common editing engine supports.

GME has both a metamodeling environment as well as a metamodel transformer that generates a new modeling environment from the metamodels. However, until recently there was a lack of generic tools to automatically generate domain-specific model transformers. Each model transformer was written by hand and was the most time consuming and error-prone phase of the MIC approach. There was a need to develop methods and tools to automate and thus, speed up the process of creating model transformers.

The MIC approach described above has gained significant attention recently with the advent of the Model Driven Architecture (MDA) by Object Management Group (OMG) [2]. MIC can be considered as a particular manifestation of MDA, which is tailored towards system construction via domain-specific modeling languages [6].

Comparison Of Features

Most MIC implementations support the specification and implementation of syntax and visualization of domain-specific languages. However, support for static semantics and dynamic semantics is not adequate. Dynamic semantics are usually represented using a general purpose programming language. This makes the code complex and difficult to maintain. Atom³ is the only system that provides a graph

grammar based transformation specification language. The language can be used for simple transformations but is not suited for complex transformations.

Table 1: Comparison of Various MIC Tools

		DOME	Moses	Atom³	KMF	MetaEdit	GME
Metamodeling language		Proprietary	GTDL (textual)	ER Diagrams	UML 1.3	Dialog box based interface	Stereotyped UML
Syntax	Vertex Type	Supported	Supported	Supported	Supported	Supported	Supported
	Edge Type	Supported	Supported	Supported	Supported	Not Supported	Supported
	Attributes	Dialog box	Textual Notation	Dialog box	XMI	Dialog box	Attribute objects composed in vertex
	Composition	Sub-diagram attribute	Child, an attribute of parent	Not supported	XMI	Supported	UML Composition
	Aggregation	Via Node Attributes	Using an edge	As a relation	XMI	Not Supported	Class with stereotype <<Set>>
	Inheritance	Using GenSpec	Not supported	Not supported	XMI	Using ancestor attribute	UML inheritance
	Reference	Not supported	Not supported	Not supported	Not supported	Not supported	Class with stereotype <<Reference>>
Visualization	Vertex	Set of predefined shapes	Graphical editor	Graphical editor	?	Graphical editor	Bmp files or specification via code
	Edge	Set of predefined line types	Set of predefined line types	Set of predefined line types	?	?	Set of predefined line types
	Attributes	embedded in the visual vertex notation	Textual notation	Dialog Box	?	embedded in the visual vertex notation	Predefined menu and/or programmable via an API
Static Semantics	Specification	Alter code	Not Supported	Not Supported	OCL	Proprietary Rule language	OCL
	Enforcement	Alter compiler	Not Supported	Not Supported	?	Proprietary	OCL evaluator
	Enforcement method	?	Not Supported	Not Supported	?	?	Event driven
Dynamic Semantics	Specification Notation	Textual	Textual	Graphical	?	Textual	Textual
	Specification Language	Alter code	Java code	Graph Grammar	?	Java	C++
	Implementation	Alter compiler	Java compiler	Converted to python	?	Java compiler	C++ compiler

Summary of Model-Based Solutions

This Chapter reviewed various techniques and associated tools used in the automation of the development of a large number of software systems.

Generative Programming (GP) consists of a variety of techniques used for the automated development of product families. GP techniques have the following features (1) Capturing the commonalities and the variabilities in the product families, (2) Development of a pool of components that can be reused within the family and possibly across families and (3) A method for the specification of the assembly. The largest effort in these techniques centers on the development of the reusable assets and is the most time consuming step.

Model Integrated Computing (MIC) on the other hand has the philosophy of developing domain-specific languages for each domain. Thus MIC tool suites comprise tools that facilitate the development of languages. This consists of developing the abstract syntax, concrete syntax, visualization, static semantics and dynamic semantics. The success of the MIC approach depends on the cost incurred in the development of a new language. Most implementations have good abstractions to capture the abstract syntax and concrete syntax of the new language. However, static and dynamic semantics are usually captured as large and often complex model interpreters. This is the bottleneck of MIC and needs to be overcome in order to have a greater impact on the software development community.

To enhance the development of translators that provide dynamic semantics we need a way to precisely specify the operation of these translators on categories of models, and to then generate code that would perform the translation. However, this task is non-

trivial as a model transformer can be required to work with two arbitrarily different domains and perform fairly complex computations. Hence, the specification language needs to be powerful enough to cover diverse needs and yet be simple and usable.

From a mathematical viewpoint, models in MIC are graphs, to be more precise: vertex and edge labeled multi-graphs, where the labels denote the corresponding entities (i.e. types) in the metamodel. Using graph theoretic results to solve this problem appears to be a possible solution and will be discussed in details in the following section.

Graph Grammars And Transformations

Graph transformations and grammars have been an active topic of research for well over 20 years. This research can be classified into two broad categories. The first category, graph grammars, is an extension of textual grammars and it gave rise to node replacement grammars [54][55] and hyperedge replacement grammars [59][60]. The second category, graph transformations, researches various mathematical fields such as category theory, set theory and algebra and extended it to graphs. The prominent works in this area are double pushout [63], single pushout [64] and programmed structure replacement systems [65]. The prominent graph transformation tools are AGG [69] and PROGRES [68].

This section is organized into sub-sections where each sub-section covers a particular class of graph grammars or transformation systems. The first sub-section discusses node replacement grammars, followed by hyper-edge replacement grammars. The next sub-section deals with algebraic approaches while the final sub-section discusses programmed graph replacement systems.

Node Replacement Graph Grammars

Node replacement grammars [54][55] are a class of graph grammars that are based primarily upon the replacement of nodes in a graph. The basic production of every node replacement grammar has an LHS subgraph (called mother graph) that produces an RHS subgraph (called daughter graph). Usually the LHS subgraph consists of only one node. The nodes that appear in the LHS of a production are similar to non-terminals in Chomsky's Grammar. The production also has a gluing construct that defines how the daughter graph will glue to the rest of the graph. Edges in this grammar formalism are usually not considered to be first class objects, i.e. they are referred by means of the nodes to which they connect. The gluing constructs distinguish one node replacement grammar from another [54].

Node Label Controlled (NLC)

NLC is one of the first node replacement grammars that appeared in literature. NLC is defined as 5-tuple [54].

$$G = (\Sigma, \Delta, P, C, S) \text{ Where}$$

- Σ - the entire alphabet set (all possible node labels)
- Δ - the alphabet set of terminals (node labels that do not appear on the LHS of any production)
- P - the set of productions
- C - the connection relationships (the gluing conditions)
- S - the start graph

Each production is defined as a non-terminal node producing a graph with terminals and non-terminals along with a set of connection instructions. For example in Figure 6 we see a production with X in the LHS and a subgraph on the RHS along with a connection relation in the box. The semantics of the production is to first delete the LHS from the graph; in this case X is deleted from the graph. Next, the RHS graph is added. Then the connections between the input graph and the newly added daughter graph are established. A connection relation is a pair of node labels of the form (l, m). If the LHS node was adjacent to a node labeled 'l' then all nodes in the RHS with label 'm' will be adjacent to the 'l' labeled node. For example, in Figure 6 the relation (c, a) implies that each 'a' labeled node in the RHS will be adjacent to any 'c' labeled neighbor of X [54].

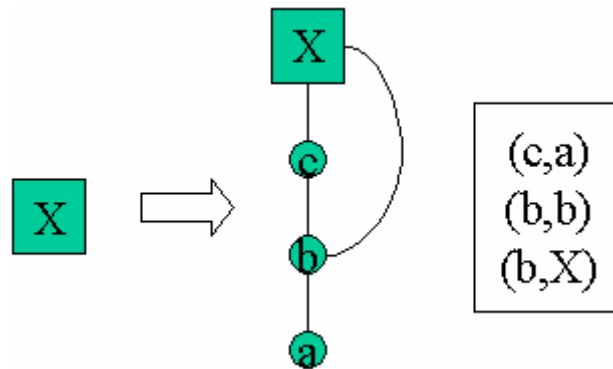


Figure 6 A NLC production

To make the production more clear let us see an example. In Figure 7 we see that (a) is the starting graph. If we apply the production denoted in Figure 6, the first step is to remove the LHS of the production, in this case X, then add the daughter graph. The result is shown in (b). Next we add the edges between the daughter graph and rest of the graph according to the gluing condition to produce the final graph shown in (c) [54].

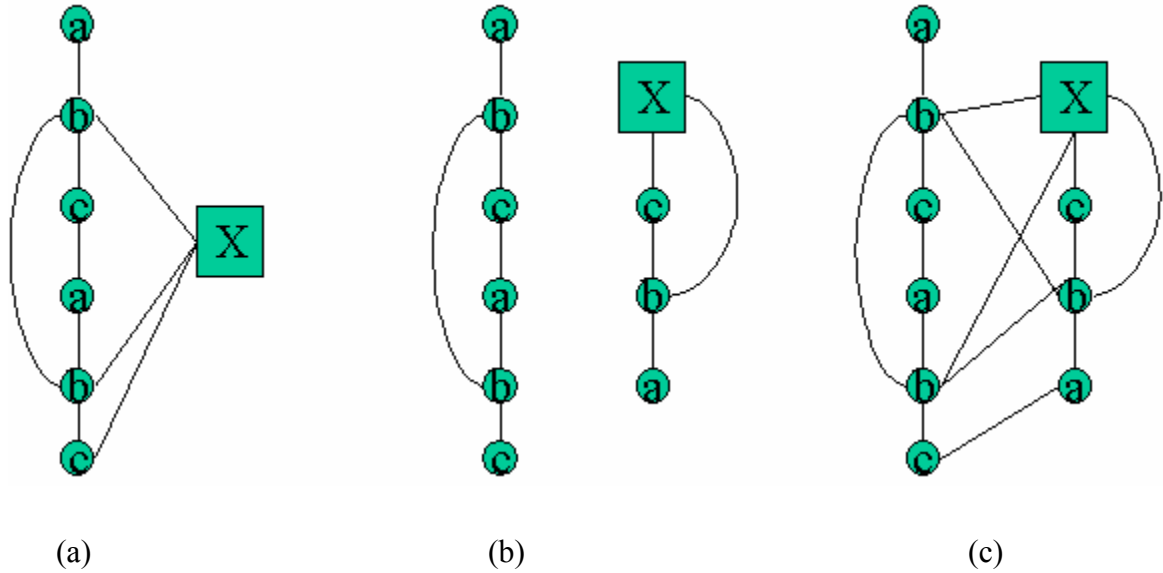


Figure 7 Application sequence of a production

In NLC, connections are made between node labels. Sometimes it is desirable to refer to a particular node while specifying the gluing construct. This gives rise to a variation where the connection relationship is of the form (u, x) , where u is a node label and x is a particular node in the daughter graph. This variation is called Neighborhood Controlled Embedding (NCE) and is studied in greater depth in the next section [54].

Neighborhood Controlled Embedding (NCE)

The formal definition of NCE is as follows

$$G = (\Sigma, \Delta, P, S)$$

Where, Σ, Δ, S are as defined for NLC and P is defined as a production of the form $P :- X \rightarrow (D, C)$ where $X \rightarrow D$ is the production and C is the connection relationship of the form (u, x) where, u is a node label and x is a particular node in the daughter graph.

NCE can be extended with direction on the edges. This gives rise to dNCE. Adding labels to the edges and allowing the connection relationship to use edge labels

makes the grammar eNCE. If both of the above mentioned extensions are added then we get edNCE which is the most popular node replacement grammar. The NCE production for the previous example is shown in Figure 8. The production states that X should be replaced by the daughter graph shown within the box. The connection relations state that every ‘c’ adjacent to X in the mother graph will be adjacent to the node ‘a’ in the daughter and every ‘b’ adjacent to X in the mother graph will now be adjacent to both ‘b’ and ‘X’ in the daughter graph [54].

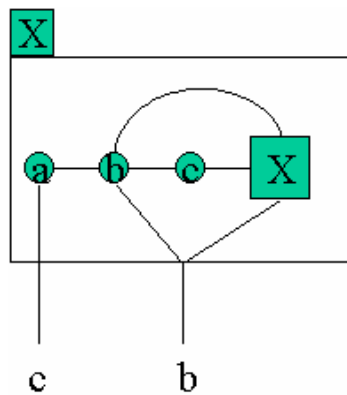


Figure 8 A NCE production

In an edNCE system the order of application of the production rules is unspecified. Thus different application sequences may lead to different graphs [54].

For this reason a property called confluence [56] is defined. A NCE grammar is said to be confluent if the output graph is the same irrespective of the sequence in which the productions are applied. Such grammars are called C-NCE. Confluence is a very interesting property as it guarantees the determinism of the productions. One possible way of achieving confluence is to have “Boundary” restriction: if all the productions in a NCE grammar do not have two non-terminals connected with an edge, and if this

property is preserved in the initial graph then, the grammar is guaranteed to be confluent. Such grammars are called B-NLC. The boundary condition extends to edNCE grammar as well [54].

The formal definition of edNCE is as follows:

Let Σ be an alphabet set of all node labels and Γ an alphabet of all edge labels.

Then a graph over Σ and Γ is defined as a tuple $H = (V, E, \lambda)$, where

V is a finite set of nodes, $E \subseteq \{(v, \gamma, w) \mid v, w \in V, v \neq w, \gamma \in \Gamma\}$ and

$\lambda : V \rightarrow \Sigma$ is the node labeling function.

The components of H are denoted as V_H, E_H and λ_H

A production of edNCE grammar is of the form

$X \rightarrow (D, C)$ where

X is a non-terminal node label,

D is a graph and

C is a set of connection instructions. $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_H \times \{\text{in}, \text{out}\}$

for readability $C = (\sigma, \beta / \gamma, x, d)$ which means that if the mother is adjacent to a node labeled σ with an edge label β then node x of the daughter will be adjacent to the node labeled σ with edge labeled γ and direction d

Let $\Sigma = \{X, Y, a, b, \sigma, \sigma'\}$ and $\Gamma = \{\alpha, \alpha', \beta, \beta', \gamma, \gamma', \gamma_1, \gamma_2, \delta, \delta'\}$.

In NCE an important concept is that of composition of embeddings. It is defined as follows: Let $P1: X1 \rightarrow (H, C_H)$ and $P2: X2 \rightarrow (D, C_D)$ be two productions where, $C_D = \{(\sigma, \gamma / \delta, y, \text{out}), (\sigma', \gamma' / \delta', y, \text{in}), (Y, \beta / \gamma_1, y, \text{in}), (Y, \beta / \gamma_2, y, \text{in}), (a, \alpha / \alpha', y, \text{out})\}$ and $C_H = \{(\sigma, \beta / \gamma, u, \text{in}), (\sigma, \beta / \gamma, u, \text{out})\}$. If H and D are disjoint graphs and the substitution of v by (D, C_D) in (H, C_H) is denoted by $(H, C_H) [v / (D, C_D)]$ embedding such that the embedding has the following property.

$$\begin{aligned}
V &= (V_H - \{v\}) \cup V_D, \\
E &= \{(x, \gamma, y) \in E_H \mid x \neq v, y \neq v\} \cup E_D \\
&\quad \cup \{(w, \gamma, x) \mid \exists \beta \in \Gamma : (w, \beta, v) \in E_H, (\lambda_H(w), \beta / \gamma, x, in) \in C_D\} \\
&\quad \cup \{(x, \gamma, w) \mid \exists \beta \in \Gamma : (v, \beta, w) \in E_H, (\lambda_H(w), \beta / \gamma, x, out) \in C_D\} \\
\lambda(x) &= \lambda_H(x) \text{ if } x \in V_H - \{v\}, \text{ and } \lambda(x) = \lambda_D(x) \text{ if } x \in V_D \\
C &= \{(\sigma, \beta / \delta, x, d) \in C_H \mid x \neq v\} \cup \{(\sigma, \beta / \gamma, x, d) \mid \exists \gamma \in \Gamma : (\sigma, \beta / \gamma, v, d) \in C_H, (\sigma, \gamma / \delta, v, d) \in C_H\}
\end{aligned}$$

The property establishes the requirements for confluence of the productions and is used to develop variants of the NCE that are confluent. A confluent grammar is one where the order of execution of the productions does not affect the outcome. Some examples of confluent grammars are boundary NLC and linear edNCE.

The embedding semantics is shown in Figure 9 where the node in question (X) is replaced by the entire graph (D, CD). Then all edges that X was associated with are substituted according to the edge substitution CD.

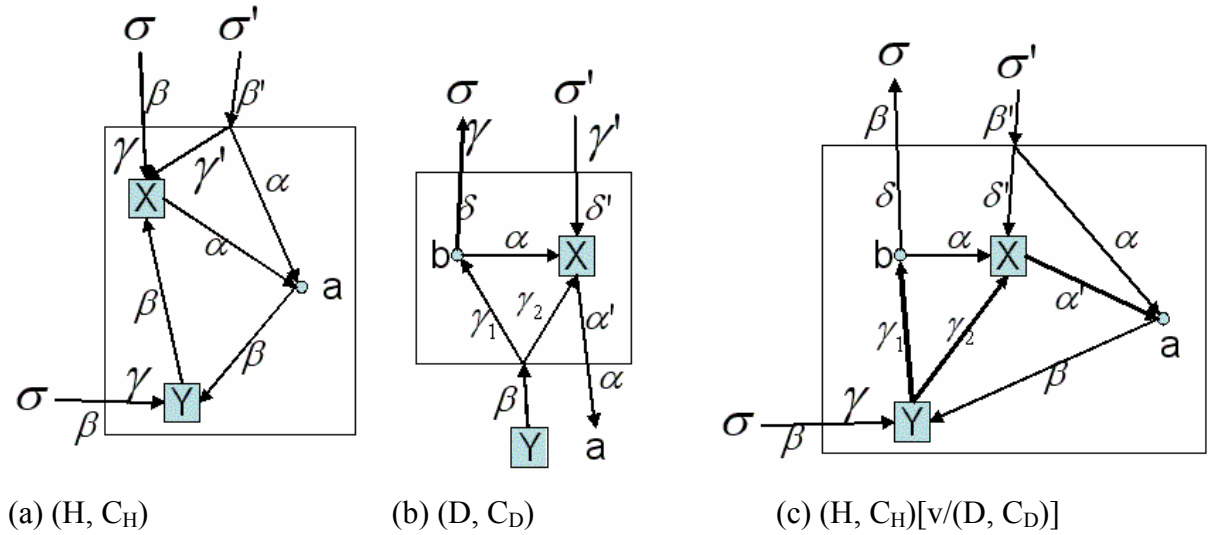


Figure 9 Two graphs with embedding and the result of their substitution [54]

The above definition of embedding has been proven to be associative in [57], thus $K[w/H][v/D] = K[w/H[v/D]]$. Another important property of composition of productions is confluence which was proved in [58].

Hyperedge Replacement Graph Grammars

Hyperedge replacement graph grammars [59][60] represents the next class of grammars that was studied. The basic philosophy is to replace an edge/hyperedge with a subgraph. For example in Figure 10 the edge e is to be replaced by the graph on the RHS.

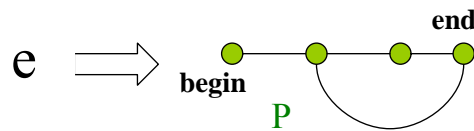


Figure 10 Hyperedge production

This production when applied to the graph in Figure 11(a) will remove the edge e from graph (a) and insert the graph from Figure 10 to produce the graph in Figure 11(b).

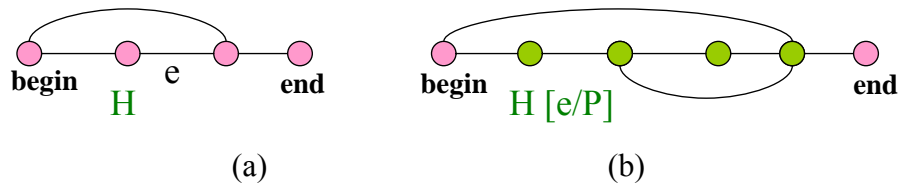


Figure 11 An example to demonstrate the hyperedge production

In general the edge to be replaced can be a hyperedge having more than two ends. Hyperedge replacement grammars have some interesting properties. The first is

sequentialization and parallelization, which states that the productions can be either applied in a sequence or all at once. This property holds true because each production works on a different edge and thus there can be no interference between them. The next property is that of confluence and it states that the order of execution of the production does not affect the result. Finally, associativity also holds true which states that if a production P1 is applied and then another production P2 is applied to the result will yield the same result as the case where P2 is applied first and P1 is applied on the result. These properties are formally defined as follows:

Sequentialization and Parallelization. Let H be a hypergraph with distinct $e_1, e_2 \dots e_n \in E_H$ and let H_1, H_2, \dots, H_n be hypergraphs. Then

$$H[e_1 / H_1, e_2 / H_2, \dots, e_n / H_n] = H[e_1 / H_1][e_2 / H_2] \dots [e_n / H_n]$$

Confluence. Let H be a hypergraph with distinct $e_1, e_2 \in E_H$ and let H_1, H_2 be hypergraphs. Then

$$H[e_1 / H_1][e_2 / H_2] = H[e_2 / H_2][e_1 / H_1]$$

Associativity. Let H, H_1, H_2 be hypergraphs $e_1, e_2 \in E_H$ Then

$$H[e_1 / H_1][e_2 / H_2] = H[e_2 / H_2][e_1 / H_1]$$

Many properties of hyperedge replacement grammars have been proven. A list of the theorems is given below:

1. **Context-freeness lemma;** it states that hyperedge replacement grammars are context free [59].
2. **Fixed-point theorem;** it states that hyperedge replacement grammars are the least fixed points of their generating productions [61].
3. **Pumping lemma generalization;** it states that each hyperedge replacement language can be decomposed into three hypergraphs FIRST, LINK and LAST such that all

sentences of the language can be constructed by a suitable composition of FIRST, k samples of LINK and LAST for each natural number k [59].

4. **Parikh's theorem;** the theorem was originally for context free languages and has been extended to hyperedge replacement grammars. It states that for all hyperedge replacement languages L and every Parikh mapping m , the set $m(L)$ is semilinear [62].

There are other interesting results based upon the kind of string languages hyperedge replacement can generate and the kind of NP-complete graph languages generated from them.

Algebraic Approach to Graph Transformation

The next approach to graph grammars is the algebraic approach [63][64]. The idea was to generalize Chomsky grammars from strings to graphs. The main aim was to come up with a generalization of string concatenation to a gluing construction of graphs. The approach is algebraic because graphs are considered a special kind of algebra and the gluing is defined by algebraic constrictions called pushouts. The pushout approach has been taken from a more general field of category theory and has been applied to the more specific field of algebraic theory of graph grammars. There are two basic algebraic approaches, (a) Double PushOut (DPO) [63] and (b) Single PushOut (SPO) [64] These approaches will be covered in their respective subsections.

To define an algebraic approach to graph grammars, first graphs have to be defined, then graph isomorphism and finally graph replacements have to be defined.

A graph is defined as two sorted algebras where the set of vertices V and the set of edges E are the carriers, and unary operations such as source $s: E \rightarrow V$ and destination

$d: E \rightarrow V$ define a relation between vertices and edges. There are labeling functions $lv: V \rightarrow L_V$ and $le: E \rightarrow L_E$, where L_V and L_E are the node and edge alphabet set.

A production $P: L \rightarrow R$ defines a partial correspondence between each element of its left and right hand side determining which nodes will be preserved, deleted or created. The first step of applying a production is to match the LHS of the production in the host graph. A match $m: L \rightarrow G$ is a graph homomorphism mapping nodes and edges of L to G , preserving the graph structure. Once a match is found for L in the LHS of a production, the next step is to remove from G all nodes and edges that have no correspondence in the RHS. Similarly those nodes and edges in R and not in L are added to G [54].

Double Pushout (DPO)

The basic approach in double pushout is to start with a subgraph to match in the input graph. Then an inverse gluing condition is applied followed by a gluing. Put simply, two graphs are drawn. The production is represented as $P \Rightarrow L \leftarrow K \rightarrow R$. After the subgraph L is matched in the host graph the first step is to remove parts of the matched graph that correspond to the elements in L and not in K . L is the inverse gluing condition and specifies what to delete from the graph. Next portions that are in R and not in K are added to the graph. This is the gluing condition. For example in Figure 12 we see that if we have a client (c) performing a job, it can stop the job, raise a request and then start performing the job again. In DPO we would say that when we find a client with a job, the production would first remove the job and then add a request and a job.

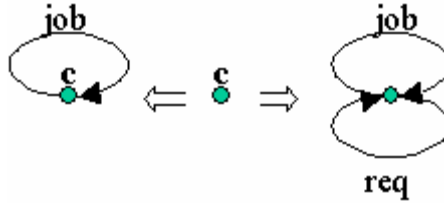


Figure 12 DPO production

In the algebraic approaches, concepts such as parallel application of productions have been defined. Parallelism in this approach can be defined in two ways: (1) based on the sequential processor model and is defined as a set of productions where the order of application doesn't affect the result. (2) a truly parallel definition where the productions are applied in parallel using one processor per production. The first approach is called *sequential independence* while the second is *explicit parallelism*. Two productions are said to be *sequentially independent* if they are not causally dependent and two productions are parallel if they are mutually exclusive.

For *explicit parallelism* there is a need to define means that will facilitate the truly parallel application of the productions. Two approaches have been suggested. The first approach is called *amalgamation*, which specifies that if there are two productions P1 and P2 then the amalgamated production $P1 \oplus_{p_0} P2$ should be present such that the production P1 and P2 can be applied in parallel and the amalgamated production P0 that represents the common parts of both the productions should be applied once. The control of application of productions has also been studied. Productions can be arranged into sequential and parallel flow.

There is another approach to parallelism in which the graph G is broken down into two parts and distributed to different processors $G_1 \oplus_{G_0} G_2$ and then the common part G_0 is used to put them together.

Single Pushout (SPO)

SPO specifies only one pushout that performs both the addition and deletion of nodes and edges. This causes ambiguity when two pattern nodes are mapped to the same node in the host graph and one pattern node is preserved in the pushout while the other is not. In SPO higher precedence is given to the deletion and thus in such cases the node will be deleted as shown in Figure 13.

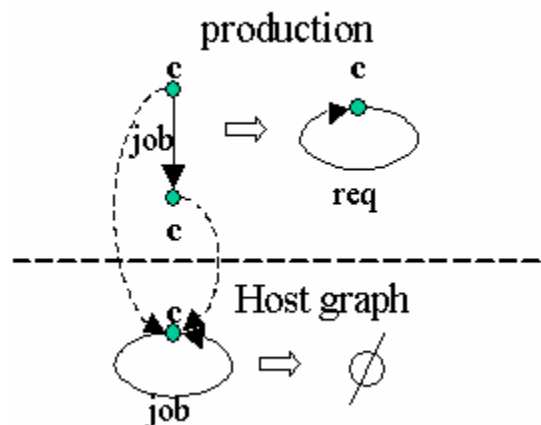


Figure 13 SPO production and example

Programmed Graph Rewriting Systems

The last section in this Chapter is on programmed rewriting systems [65]. These represent the set of practical rewriting systems and have more working implementations than theorems. The pioneers in this field are the developers of PROgrammed GRaph REplacement System (PROGRES [68]).

This section is broken down into three sub-sections. The first discusses graph replacement systems. The second deals with programmed graph replacement systems and how to apply control flow mechanisms to the graph replacement system. The third sub-section discusses PROGRES's approach to programmed graph rewriting.

Logic-Based Structure Replacement System

This section [65] critiques graph replacement system and points out their deficiencies. The problems with algebraic and other such graph replacement systems in practice are the following:

1. A lack of static integrity constraints on graphs
2. Specification of derived attributes and relations
3. The implicit use of depth first search and backtracking.

These lead to the development of PROGRES that addresses the problems stated above. PROGRES uses structure replacement as its mathematical model to define graph replacements. Graphs are defined as structures with certain properties. In order to provide integrity constraints, a *signature* is defined

A signature is defined as a 5-tuple

$\Sigma := (A_F, A_P, \nu, \omega, \chi)$ where

A_F is an alphabet of function symbols

A_P is an alphabet of predicate symbols

ν is a special alphabet of object identifier constants

ω is a special alphabet of constants representing sets of objects

χ is an alphabet of logical variables used for quantification purpose

In structure replacement systems, graph semantics have to be defined. Below is the definition of a class of graphs that show people, their income and relationship to other people

$$A_F := \{child, woman, wife, man, person, income, int, 0, \dots, + \}$$
$$A_P := \{node, edge, attr, type, \dots\}$$
$$v := \{He, She, Value, \dots\}$$
$$\omega := \{HerChildren, HisChildren, \dots\}$$
$$\chi := \{x_1, x_2, \dots\}$$

A_F defines symbols for node labels, edge labels, attribute types, attribute values and evaluation functions.

A_P consists of four predicate symbols that define this structure to be a graph

- Node(x,l): graph contains a node x with label l
- Edge(x,e,y): graph contains edge, labeled 'e' that is incident on x and y
- Attr(x,a,v): attribute a at node x has value v
- Type(v,t): attribute v has type t

V is a set of arbitrarily chosen constant names that are used to refer to single objects in the graph while W is a set of names used to refer to a set of objects matched in the graph. X is a set of names used for quantification purposes.

A structure is defined as a set of formulas. For example a structure of the person database can be $F := \{node(Adam, man), node(Eve, woman), node(Sally, woman), \dots, attr(Adam, income, 5000), attr(Eve, income, 10000), \dots, edge(Adam, wife, Eve), edge(Eve, child, Sally), \dots\}$

A schema defines all the possible structures that are legal. It is defined as a set of implications and equivalences. For example,

$$\begin{aligned}
\Phi := & \{ \forall x, e, y : \text{edge}(x, e, y) \rightarrow \exists xl, yl : \text{node}(x, xl) \wedge \text{nde}(y, yl), \\
& \forall x, a, v : \text{attr}(x, a, v) \rightarrow \exists l : \text{node}(x, l), \dots \} \\
\cup & \{ \forall x : \text{node}(x, \text{man}) \rightarrow \text{node}(x, \text{person}), \\
& \forall x : \text{node}(x, \text{woman}) \rightarrow \text{node}(x, \text{person}), \\
& \forall x, y : \text{edge}(x, \text{wife}, y) \rightarrow \text{node}(x, \text{man}) \wedge \text{node}(y, \text{woman}), \\
& \forall x, y, z : \text{edge}(x, \text{wife}, y) \wedge \text{edge}(x, \text{wife}, z) \rightarrow y = z, \\
& \dots \} \\
\cup & \{ \forall x, y : \text{ancestor}(x, y) \leftrightarrow (\exists z : \text{edge}(z, \text{child}, x) \wedge (z = y \vee \text{ancestor}(z, y))), \\
& \dots \}.
\end{aligned}$$

A Structure replacement rule is defined as a quadruple

$$\begin{aligned}
p := & (AL, L, R, AR) \\
AL, AR \in & F(\Sigma) \text{ where } F(\Sigma) \text{ is the set of all negative conditions} \\
L, R \in & \lambda(\Sigma) \text{ where } \lambda(\Sigma) \text{ is the set of all structures}
\end{aligned}$$

For the example in Figure 14 the content of these sets is:

$$\begin{aligned}
L := & \{ \text{node}(\text{He}, \text{man}), \text{node}(\text{She}, \text{woman}), \\
& \text{attr}(\text{He}, \text{income}, \text{HisValue}), \text{attr}(\text{She}, \text{income}, \text{HerValue}), \\
& \text{edge}(\text{He}, \text{child}, \text{HisChildren}), \text{edge}(\text{She}, \text{child}, \text{HerChildren}) \} \\
AL := & \{ \neg \text{brother}(\text{She}, \text{He}), \neg \exists x : \text{edge}(\text{He}, \text{wife}, x), \neg \exists x : \text{edge}(x, \text{wife}, \text{She}), \\
& \text{HisValue} < \text{HerValue}, \\
& \text{node}(\text{HisChildren}, \text{person}), \text{node}(\text{HerChildren}, \text{person}) \} \\
R := & L \setminus \{ \text{attr}(\text{She}, \text{income}, \text{Value}) \} \\
& \cup \{ \text{edge}(\text{He}, \text{wife}, \text{She}), \text{attr}(\text{She}, \text{income}, \text{Value} + \text{tax Reduction}(\text{Value})), \\
& \text{edge}(\text{He}, \text{child}, \text{HerChildren}), \text{edge}(\text{She}, \text{child}, \text{HisChildren}) \} \\
AR = & \{ \}
\end{aligned}$$

In simple words, a replacement production consists of a LHS subgraph, a RHS subgraph, a guard and attribute mapping constructs. The subgraphs have concepts such as negative edges, sets of nodes, optional nodes and edges.

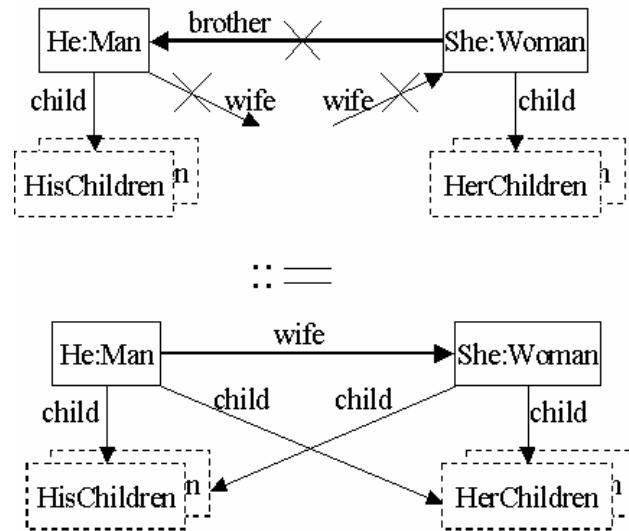


Figure 14 A production in the PROGRES system

The example production in Figure 14 demonstrates most of the language features. The production specifies a pattern containing a man and woman vertex such that they are not brother/sister and are not married. The RHS of the production create an edge called wife from woman to man. In the production, brother is a negative edge in the LHS. The children nodes are dashed which means they are optional and the matcher should match 0..* children for the given parent and thus showing the optional feature and the feature to match a set of nodes. The RHS of the rule specifies that a wife edge should be added and child edges should be added to the matched pattern. Apart from the LHS and RHS, there are constraints, which should evaluate to true for the rule fire. PROGRES also has a language to specify attribute mapping of the patterns.

Programmed Structure Replacement Systems

This section deals with the organization of the rules. In traditional graph grammars the execution semantics for rule execution is defined based on the availability

of the LHS in the graph. Other approaches deal with issues such as production priority and regular expressions to specify rule fire sequences and control flow graphs. The author identified some desirable characteristics of the control flow. They are:

1. **Boolean nature:** Application of a transformation should result in either success or failure.
2. **Atomic character:** A sequence of replacement steps should modify the graph if and only if all of its intermediate steps succeed.
3. **Consistency preserving:** The replacements have to preserve the consistency as specified by separately defined integrity constraints.
4. **Nondeterministic behavior:** A single rule replaces any match of its LHS.
5. **Recursive definition:** Transformations should be allowed to call other transformations without restrictions.

To fulfill these criteria PROGRES uses operators defined by Dijkstra in [66] and extended by Nelson in [67] to produce a formal language that can be verified using proofs. The constructs used are:

- Skip – Always returns true and relates a given graph to itself.
- Loop – will either loop forever or crash.
- Def(a) – an action that succeeds if a returns true.
- Undef(a) – an action that succeeds if a returns false.
- (a ; b) – a sequential execution of a followed by b.
- (a | b) – a nondeterministic choice between a and b.
- (a & b) – returns the intersection of results of a and b.

Summary of Graph Grammars and Transformations

This section discussed the various graph grammars and transformations approaches published in literature. These include node replacement grammars, hyperedge replacement grammars, algebraic approaches, and programmed graph replacement systems.

Graph grammar techniques such as node replacement and hyperedge replacement grammars are direct extensions of textual grammars and are well suited for the specification and recognition of graphical languages. In textual languages, grammar is used primarily for parsing raw textual streams into tree data structures. Unlike textual languages, graphical languages are built with a database/data-structure backend and do not require a parsing phase. Grammars have the ‘execute when LHS sub graph found’ semantics: whenever a pattern is found in the host graph the particular rule in question will fire. This brings about two different issues. The first is that of confluence: the effect of the execution of the rules in different orders and the second is efficiency. Sub-graph isomorphism is an NP complete problem and thus the time complexity of the implementations is also a concern.

Graph transformations take a different approach than that of grammars. Here the focus is on the transformation of a graph, including addition/deletion and modification of the graph. Algebraic approaches such as single and double pushout are transformation methods. They define transformations as algebraic constructs and take care of confluence by the use of sequencing of rules. However, the sequencing constructs are primitive and not adequate for specifying complex transformations. These transformation languages do not provide traversal strategies. The most mature of the transformation systems is the

Programmed Structure Replacement System (PSRS) which uses the structure replacement as the basis of the transformation. On top of the transformation there is a high-level control flow language for the explicit sequencing of rules. PSRS also has a few drawbacks as a language for model-to-model transformation. This language was developed to perform manipulations on databases and can only perform graph manipulations within the same domain.

Graph Transformation Based Tools

The theory described above has given rise to many tools. Prominent amongst these tools are PROGRES [68], AGG [71], DiaGen [82], GenGEd [83], VIATRA [84]. Out of these tools PROGRES, AGG and VIATRA support general purpose graph transformation languages while DiaGen and GenGEd are visual language environments.

PROGRES

Programmed Graph Replacement System (PROGRES) is a tool developed at Lehrstuhl für Informatik III, University of Technology Aachen (RWTH Aachen), Germany.

PROGRES consist of an editor for the specification of the graph domain. The domain/type system is defined using a proprietary textual language called Schema. Schema has advanced type concepts such as inheritance, composition, implicit and explicit attributes [65][68].

Transformations are specified in Programmed Structure Replacement, a language with control flow semantics on top of the graph transformation. Graph transformations

are specified using a graphical editor and can be embedded in the textual control flow of the transformation [65][68].

AGG

The domain tools of AGG consist of a graphical editor for the specification of type graphs. Type graphs are the language used to specify the type system for the domain. Users can create node and edge types and specify the type requirement for the source and destination of the edges. The type system is simplistic and lacks concepts such as composition and inheritance. The type system does not have support for semantic constraints, that are based on attributes. Furthermore, a project can have only one type graph. Type checking based on the type system can be enabled and/or disabled very easily. There is limited support for the specification of the visualization features of vertices and edges [69][70][71].

The graph tools of AGG consist of a graphical editor for the specification of the graphs. Graphs can be created using types defined in the type graph. Alternatively, the user can disable the type graph and define node/edge types while creating the host graph [71].

The transformation tools include a visual transformation specification editor and a transformation engine that can perform the transformation. The transformation language used is single pushout. The visual editor allows the users to create and execute the rules. Additionally, a Java API is also provided that can be used to perform the transformation. A proprietary XML format called ggx is used to store graphs, transformation rules and the type graph [71].

Comparison of Features

A set of requirements was created and it was used as the basis of comparison. The feature set chosen for the comparison is divided into three groups. (1) Domain Specification: A set of features required to describe and enforce the graph domain. (2) Graph Specification: A set of features required to specify graphs in each tool and (3) Transformation Specification: A set of features required to specify and execute the transformations.

Table 2: Comparison of Graph Transformation Tools

		AGG	PROGRES	ATOM³
Domain Specification	Language	Type graph	Schema	ER-diagrams
	Notation	Graphical	Textual	Graphical
	Inheritance	----	Supported	Not supported
	Attribute Types	Supported	Supported	Supported
	Constraints	Not Supported	Not Supported	Not supported
	Multiple domains	Not Supported	Not Supported	Supported
Graph Specification	Format	Ggx format	GRASS database	
	Editing method	Graphical	Database manipulation	Graphical
	Composition	Not supported	Not supported	Not supported
	Domain enforcement	Optional	By the database	By visual editor
Transformation Specification	Language	Single Pushout	Programmed Structure Replacement System	Graph Grammar
	Notation	Graphical	Textual & graphical	Dialog based Graphical
	Pattern Specification	Single cardinality	Single Cardinality	Single Cardinality
	Between Domains	Not Supported	Not Supported	??

Critique of Graph Transformation Tools

Graph grammar techniques such as node replacement grammars, hyperedge replacement grammars, and algebraic approaches such as the ones used in AGG do not provide sufficiently expressive mechanisms for controlling the application of

transformation rules. PROGRES has a rich set of control mechanisms; however, they only perform transformations within the same schema. Schema [68] in PROGRES and type graphs [71] in AGG can be considered as a graph grammar that specifies a family of allowed graphs. If semantics is assumed for the types in these schemas/type graphs then they considered as the specification for a domain. These specifications capture structural and integrity constraints that the graphs must conform to. In both PROGRES and AGG the transformations can only be written such that the graph conforms to a single domain specification. That is, the graph must at all times conform to the singleton schema/type graph. Transformations that convert a graph belonging to one schema/type graph to another one conforming to a different schema/type graph are not possible in these systems.

In MIC, a domain is represented by a metamodel, and the model transformations typically transform models/graphs that conform to one metamodel to models/graphs that conform to a completely different metamodel. For example, a model transformer may be required to convert models/graphs belonging to the “state machine” domain to models/graphs conforming to the “flow chart” domain. The graph transformation system must provide support for these transformations across heterogeneous domains. There is yet another problem: maintaining references between the different models/graphs. During the transformations it is usually required to link graph objects belonging to different domains.

To illustrate the point let us consider a very simple transformation that needs to transform models conforming to one domain to another. For sake of simplicity let the source domain have one vertex type $V1$ and one edge type $E1$. Similarly, the target

domain has one vertex type $V2$ and one edge type $E2$. The transformation's aim is to create a vertex in the target for each vertex in the source and an edge in the target corresponding to each edge in the source such that:

$$\forall e1 \in E1 \Rightarrow \exists_1 e2 \in E2, \forall v2 \in V1 \Rightarrow \exists_1 v2 \in V2 \quad \textbf{Relation 1}$$

A simple algorithm could first create a target vertex for each source vertex and then create the edges. To create a target edge $e2$ that corresponds to the source edge $e1$ we need to find the vertices in the target that correspond to the source vertices $e1$ is incident upon. This information needs to be saved in the first phase of the transformation for use in the second phase, and can be considered as maintaining a reference between two graphs. There are other examples where referencing is not that easy, for example, a transformation that determines the cross product of two sets of vertices to generate a new set of vertices. In this case each pair of source vertices should reference a single target vertex. A method is required to specify and use this information.

The existing graph grammars and transformations are based on powerful mathematical concepts but not well suited for the specification and implementation of model transformers as described. Hence, a new approach that targets the specific needs of model-to-model transformation is required.

CHAPTER III

RESEARCH PROBLEM, HYPOTHESIS AND METHODS

Model Integrated Computing (MIC) [1] advocates the use of domain-specific concepts to represent system design. Domain-specific models are then used to synthesize executable systems, perform analysis or drive simulations. Using domain concepts to represent system design helps increase productivity, makes systems easier to maintain and evolves and shortens the development cycle [1].

Object Management Group (OMG) has proposed the use of models as a complete specification of software artifacts. In their recent initiative called Model Driven Architecture (MDA) [2] the aim is to allow developer to model software without thinking about its implementation platform. Such a model is called a Platform Independent Model (PIM). A PIM can then be transformed into a Platform Specific Model (PSM) for a platform with the help of automated generators. The language proposed for the specification of such PIMs and PSMs is UML 2.0.

MIC can be considered as a methodology for Domain Specific MDA (DSMDA) where the focus is on developing the MDA process for specific domains. An implementation of DSMDA should consist of a Domain Specific Modeling Environment that allows users to describe systems using domain concepts. This environment is then used to develop Domain Specific Platform Independent Models (DSPIMs). These models represent the behavior and structure of the system with no implementation details. Such models then need to be converted to a Domain Specific Platform Specific Models (DSPSM). DSPSM could either be based on the use of domain-specific libraries and

frameworks or not have any domain-specific information. It is a term that covers all possible platform based models.

Domain Specific MDA however, has its own problems such as high development cost, lack of standardization, and lack of vendor support [6]. These problems can be tackled by developing a framework to support the creation and use of Domain Specific Modeling Environments (DSME). The cost of the framework is distributed over all the projects built using it. Recurring needs can be factored out and implemented once in the framework. The framework can also help in the standardization of DSME specifications, thus providing a common vocabulary and standards based interfaces for vendors.

There are a set of minimal requirements that such a framework must fulfill. In order to reduce the time required for the development of DSMEs, the framework must provide tools to speedup each aspect of DSME creation. In Chapter III the section on Model Integrated Computing (MIC) lists a set of basic features the framework should support.

Tools such as GME [14], Atom³ [15], DOME [16] and Moses [17] already provide a major portion of the framework support. Among these tools GME supports the greatest number of features (see Table 1). Currently however, dynamic semantics are specified and implemented using code. DSME developers spend a significant amount of time and energy in writing code that implements the transformation from Domain Specific Platform Independent Model (DSPIM) to Domain Specific Platform Specific Model (DPSM).

For a framework to be successful, it should significantly lower the time required to specify and implement DSMDAs. This includes the specification and implementation

of the dynamic semantics. A high-level specification language is required for the specification of model transformers. An execution framework can then be used to execute such specifications. Currently, the specification and implementation of the dynamic semantics of domain-specific languages requires significant effort and is the bottleneck in the MIC process.

To speed up the development of DSMDAs a formal methodical approach needs to be developed for the specification and automatic implementation of model transformers. Design of such a language is non-trivial as a model transformer can work with arbitrarily different domains and can perform fairly complex computations. The specification language needs to be powerful enough to cover diverse needs and yet be simple and usable.

When observed from a mathematical viewpoint, models in MIC are graphs, more precisely they are typed, attributed multi-graphs. Thus, the model transformation problem can be converted into a graph transformation problem. We can use the mathematical concepts of graph transformations [54] to formally specify the intended behavior of a model interpreter.

In Chapter II we saw that graph grammars and graph transformations have been recognized as a powerful technique for specifying complex transformations that can be used in various situations in a software development process [74][75][76][77]. Many tasks in software development can be formulated using this approach including weaving of aspect-oriented programs, [78] application of design patterns [76], and the transformation of platform-independent models into platform specific models [6].

Graph grammars have been developed mostly for the specification and recognition of graph languages while graph transformations have been developed with the intention of manipulating the source graph into the target graph. For model transformations however, graph rewriting, i.e. when a source graph is traversed to produce a second disconnected graph is required along with transformations. One of the primary differences between transformation and rewriting is that in transformation the input and the output graph both belong to the same family of graphs while in a rewriting these graphs could belong to different families. These graph families are specified with the help of different constructs. For example, PROGRES uses a language called Schema for specifying the graph family while AGG use the notion of type graphs. A family of graphs forms a domain that specifies the set of all allowable graphs a transformation/rewriting can handle. In a rewriting the domain of the input graph may be different from the domain of the output graph. In summary, the following features are required in the transformation language:

1. The transformation language should have a sub-language for the specification of graph domains.
2. The domain specification language should use a well know language or be based on one.
3. The transformation should use the type information from the domains to strongly type the transformations
4. Often rewriting graphs belonging to one domain into graphs that belong to another domain is required.
 - a. The language should support the specification of multiple domains.

- b. It should have constructs that allow users to write rewritings where the input and output graphs are disjoint and do not even belong to the same domain.
5. The computational power of the transformation language should be comparable to a Turing machine to ensure that any transformation conceivable can be handled by it.
6. The language should be capable of transforming/rewriting any number of graph/domain pairs, not just two. There could be n input graphs and m output graphs and these graphs can belong to any number of domains.
7. The language focus should be on constructs that allow users to write efficient transformations.
8. The language should have efficient implementations of its programming constructs. The implementation should be comparable to its equivalent hand written code.
9. The language should have a formal mathematical foundation that can facilitate the formal verification of transformations by theorem proving or other formal techniques.

Research Hypothesis

“A Metamodel based transformation language using graph rewriting and transformations that support multiple graphs (that may belong to different domains) with an efficient implementation is suitable for the specification of model transformers. Such a language should help shorten the time taken to develop model transformers and allow for formal proof of correctness of the transformations.”

Research Methods

The aim of the dissertation is to define a language for model-to-model transformations based on graph grammar and transformations techniques. The language development is based on identifying requirements of model transformations and then researching how these needs can be fulfilled by simple and formal constructs. Requirements were gathered by looking at various target applications and by creating a list of challenge problems. Two challenge problems were been chosen:

1. Generate a non-hierarchical Finite State Machine (FSM) [9] from a Hierarchical Concurrent State Machine (HCSM) representation similar to Statecharts [21]. This problem introduces interesting challenges. To map concurrent state machines to a single machine there is a need for complex operations that include computing the Cartesian product of the parallel state space. Evaluation of this particular transformation requires a depth-first bottom-up approach and will test whether the system can allow different traversal schemes.
2. Generate from a given Simulink/Stateflow model the equivalent Hybrid Automata [99]. This is another non-trivial example as the mapping is not a straightforward one-to-one mapping. It is not even obvious if the problem can be solved in the most general case. The algorithm used to solve this problem converts a restricted Simulink-Stateflow model to its equivalent hybrid system. This algorithm has some interesting steps such as state splitting, reachability analysis and special graph walks that make it a challenging problem to solve.

The complexity of the example problems gives confidence that if solutions to these problems can be specified in the new language and efficient code can be generated

from such a specification then the language will be expressive enough to be used to solve a large number of non-trivial real world problems.

The next step is to develop language constructs that can solve these challenge problems. Development of language constructs includes its syntax, visualization, semantics and algorithms for its execution. These language constructs should then be evaluated on the basis of expressiveness, generality and efficiency. A few candidate constructs will then be implemented in the execution engine to further evaluate them.

A large part of this research effort focused on the execution framework for the language. Initially a working, non-optimized framework was used to test the language constructs. The framework was made flexible to try out different constructs and algorithms. It was then used to test different algorithms and the execution of language constructs.

Completion Criteria

The completion criteria were that the developed language and execution framework should meet the requirements specified in this Chapter and it should be able to successfully solve the challenge problems. Two criteria were developed to measure the success of the language.

The first criterion was the expressiveness of the language. It should have all the language constructs required to specify the challenge problems and the implementation of the languages should be able to provide working solutions to them. This criterion is aimed to measure the ability of the language to express solutions to complex problems. Measures such as Turing completeness can be used to prove the computational power of the language.

The second is the usefulness of the language for solving model-to-model transformation problems. This can be argued by demonstrating language constructs in the new languages and how they may be better than the traditional approaches. User feedback can also be used to validate the theoretic claims. Measures such as the time taken to solve a problem in the new language verses hand code can also be used to further strengthen the case.

CHAPTER IV

GREAT: A MODEL-TO-MODEL TRANSFORMATION LANGUAGE

To address the requirements laid out in the dissertation proposal and to validate the research hypothesis a model-to-model transformation language was developed. The language is called Graph Rewriting And Transformation (GReAT). This language has four distinct parts:

- Heterogeneous Transformations.
- Pattern Specification language.
- Graph transformation language.
- Control flow language.

Heterogeneous Graph Transformations

Many approaches have been introduced in the field of graph grammars and transformations to capture graph domains. For instance, schemas are used in PROGRES [68] while AGG [69] uses type graphs. These approaches are specific to the particular systems and each have some features that others cannot replicate. Standards like UML are widely used in the software community today, are well understood and are able to express a super set of the constructs allowed in the other languages. For these reasons we have chosen to follow the UML route. It was also a pragmatic decision, as UML was used in the tools that were used for developing the language.

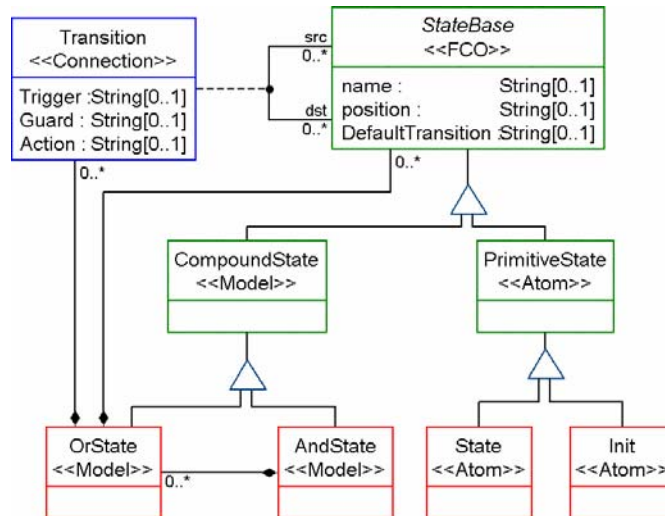


Figure 15 Metamodel of hierarchical concurrent state machine using UML class diagrams.

In model-to-model transformations the input and output graphs are object networks whose “schema” can be represented using UML class diagrams and expressions in the Object Constraint Language (OCL) [87]. UML provides a rich language to specify structural constraints while OCL can be used to specify non-structural, semantic constraints. Thus, a UML class diagram plays the role of the graph grammar as it can describe all the “legal” object networks that can be constructed within the domain. Finally, UML can also be used to generate an object-oriented API that can be used to traverse the input graph and to generate the output graph. GReAT allows the user to specify any number of domains that can be used for the transformation. Figure 15 shows a UML class diagram that represents the domain of Hierarchical Concurrent State Machines (HCSM) and Figure 16 shows the metamodel of a Finite State Machine (FSM).

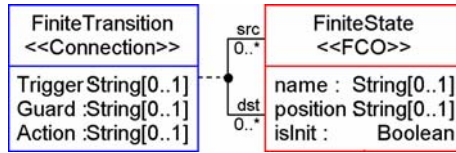


Figure 16 Metamodel of a simple finite state machine

There is yet another problem that is maintaining references between the different models/graphs. During the transformations it is usually required to link graph objects belonging to different domains.

To illustrate the point let us consider a very simple transformation that needs to transform models conforming to one domain to another. For sake of simplicity we consider that the source domain has only one type on vertices $V1$ and only one type of edges $E1$ and that the target domain has again only one type of vertices $V2$ and only one type of edges $E2$. The transformation's aim is to create a vertex and edge in the target set for each vertex and edge in the source set:

$$\forall e1 \in E1 \Rightarrow \exists_1 e2 \in E2, \forall v2 \in V1 \Rightarrow \exists_1 v2 \in V2 \quad \textbf{Relation 2}$$

(where \exists_1 means “precisely one”). A simple algorithm could first create a target vertex for each source vertex and then create the edges. To create a target edge $e2$ that corresponds to source edge $e1$ we need to find the vertices in the target that correspond to the two source vertices $e1$ is incident with. This information needs to be saved in the first phase of the transformation for use in the second phase, and can be considered as maintaining reference between two graphs. There are other examples where the referencing is not that easy, for example, in a transformation that determines the cross product of two sets of vertices to generate a new set of vertices. In this case each pair of source vertices should reference a single target vertex.

This problem was tackled in GReAT by using an additional domain to represent all the cross-domain links. Apart from using UML to specify all the different domains that will be used for the transformation, UML is also used to specify a temporary domain that contains the information of all the types of cross-links the transformation needs. For example, Figure 17 shows a metamodel that defines associations/edges between HCSM and FSM. The *State* and *Transition* are classes from Figure 15 while the *FiniteState* and *FiniteTransition* are classes from Figure 16. This metamodel defines three types of edges. There is a *refersTo* edge type that can exist between *State* and *FiniteState* and between *Transition* and *FiniteTransition*. Another edge type *associatedWith* is defined and it can exist between *State* objects.

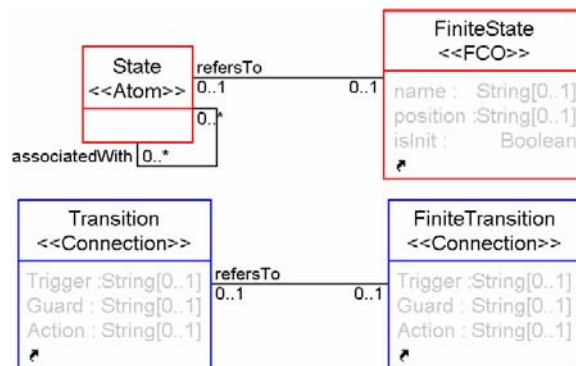


Figure 17 A metamodel that introduces cross-links

Cross-links can be defined not only between different domains but can also be used to extend a domain to provide some extra functionality required by the transformation. By using a separate domain to specify the cross-links we are able to tie the different domains together to make a larger, heterogeneous domain that encompasses all the domains and cross-references. This also helps us to have the same representation for cross-links and for domain edges.

Definitions

Before describing GReAT, some initial definitions are presented in this section. Graphs used in the GReAT language are typed and attributed multi-graphs and are defined below.

Vertex: A vertex V is a pair: (class, attrs), where class is a UML class, and attrs is a map that maps each defined attribute of the class into a value.

Edge: An edge E is a 3-tuple (etype, src, dst), where etype is the association the edge belongs to. In UML, simple associations are distinguished by their endpoint classes. This information can be considered as an “edge type”. The association classes in UML can also be distinguished using two edges: one from the source class to the association class and another one from the association class to the destination class. Src and dst are the vertices that the edge is incident upon. The class of these vertices must be identical to the endpoint classes of etype.

Graph: A graph G is pair (GV, GE) , Where GV is a set of vertices in the graph and GE is the set of edges and $\forall e = (etype, src, dst) \in GE, src \in GV \wedge dst \in GV$.

Match: A match M is a pair (MVB, MEB) , where MVB is a set of vertex bindings and MEB is a set of edge bindings. Vertex binding is defined as a pair (PV, HV) , where PV is a pattern vertex and HV is a host graph vertex. Similarly, edge binding is a pair (PE, HE) , where PE is a pattern edge and HE is a host edge. The match must satisfy the following property.

$$\begin{aligned} &\forall EB \in MEB, \text{ where} \\ &EB = (pe, he), pe = (pet, psrc, pdst), he = (het, hsrc, hdst) \\ &\exists VBS \in MVB \wedge VBD \in MVB \\ &\therefore VBS = (psrc, hsrc) \wedge VBD = (pdst, hdst) \end{aligned}$$

The match does not have any restriction that specifies that each pattern object must have a binding. This is intentional, as the match is also used to specify partial matching of pattern graphs. The default behavior of the pattern matcher is to have a unique match for every pattern vertex however there are cases when this is not desirable.

The Pattern Specification Language

A full graph transformation language is built upon a graph pattern specification language and pattern matching. Graph patterns allow selecting portions of the input (host) graph, and thus specify the scope of individual transformation steps. The specification techniques found in graph grammars and transformation languages [54][65][69][70][78][79][80][81] were not sufficient for our purposes, as they did not follow UML concepts. This section introduces an expressive yet easy to use pattern specification language that is closely related to UML class diagrams.

Recall that the goal of the pattern language is to specify patterns over graphs (of objects and links), where the vertices and edges belong to specific classes and associations. In this language we will rely on the assumption that a UML class diagram is available for the objects. The UML class diagram can be considered as the “graph grammar,” which specifies all legal constructs formed over the objects that are instances of classes introduced in the class diagram.

Simple Patterns

A simple pattern is one in which the pattern represents the exact subgraph. For example, if we were looking for a clique of size three in a graph, we would draw up the clique as the pattern specification. These patterns can be alternatively called single

cardinality patterns, as each vertex drawn in the pattern specification needs to match exactly one vertex in the host graph.

These patterns are straightforward to specify; however, ensuring determinism on such graphs is not. In this case, determinism means that given a graph and pattern the match returned should be the same from one execution of the pattern matcher to another and from one matching algorithm to another. Pattern matching in graphs is non-deterministic and different matching algorithms may yield different results.

Consider the example in Figure 18(a). The figure describes a pattern that has three vertices P1, P2 and P3, each of type T. The pattern can match with the host graph shown in Figure 18(b) to return two valid matches, $\{(P1, T1), (P2, T3), (P3, T2)\}$ and $\{(P1, T3), (P2, T5), (P3, T4)\}$. For sake of brevity matches are considered as a set of vertex bindings, edge bindings have been ignored as they can be inferred from the vertex bindings. We see that the result of the matching depends upon the starting point of the search and the exact implementation of the algorithm.

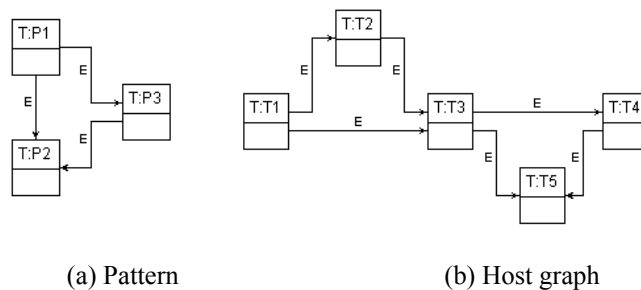


Figure 18 Non-determinism in matching a simple pattern

The solution for this problem is to return the set of all the valid matches for a given pattern. The set of matches will always be the same for a given pattern and host graph.

Returning all the matches however, has a time complexity of $O(C_h^{C_p})$, where C_h is the number of host vertices and C_p is the number pattern vertices. To make the pattern matching usable it needs to be optimized. One approach is to start the pattern matcher with an initial context. A context is used to start pattern matcher with an initial partial match. For example, in Figure 18 the pattern matcher could be started with a binding $\{(T1,P1)\}$. Thus, the context for the matching is the host vertex T1 and the matcher will return only one match $\{(P1,T1), (P2,T3), (P3,T2)\}$. The initial binding reduces the search complexity in two ways, (1) the exponential is reduced to only the unmatched pattern vertices, and (2) only host graph elements within a distance d from the bound vertex are used for the search, where d is the longest pattern path from the bound pattern vertex.

An algorithm for matching such kinds of patterns is given in Appendix A. The algorithm takes as input the pattern, host graph and a partial match and returns a set of matches. The partial match must have at least one vertex of the pattern bound to the host graph. It uses a recursive approach to solving the matching problem and returns a set of matches. There are cases where the pattern matcher has to be used on the entire graph without restricting it to a context. This can be achieved by running the pattern-matching algorithm for each host vertex.

Fixed Cardinality Patterns

If we need to specify a string pattern that starts with an ‘s’ and is followed by 5 ‘o’-s. The ‘o’ could enumerate five times and write the patter as “sooooo”. However, this is not a scalable solution and a representation format is required to specify such strings in a concise and scalable manner. For strings a notation could be devised where the pattern

is written as “s5o” and the semantic meaning of such the notation would be that o needs to be enumerated 5 times.

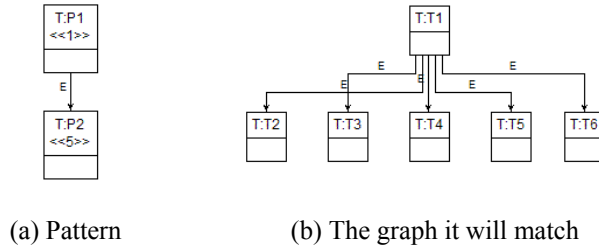


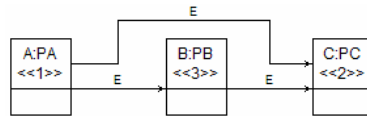
Figure 19 Pattern specification with cardinality

The same argument holds for graphs, and a similar technique can be used. The pattern vertex definition can be changed to a pair (class, cardinality), where cardinality is an integer. Vertex binding can also be redefined as a pair (PV, HVS), where PV is a pattern vertex and HVS is a set of host vertices. For example, Figure 19(a) shows a pattern with cardinality on vertices. The pattern vertex cardinality is specified in angular brackets and a pattern vertex must match n host graph vertices where n is its cardinality. In this case the match is $\{(P1, T1), (P2, \{T2, T3, T4, T5, T6\})\}$.

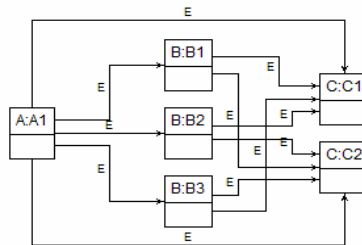
The fixed cardinality pattern and matching also have non-determinism. Even in this case the issue can be dealt with by returning all the possible matches. If all the possible matches are returned, there is a problem of returning a large number of matches. For example in Figure 19, if the host graph contained another vertex T7 adjacent to T1 then the number of matches returned would be 6C_5 (all combinations of 5 vertices out of 6). Thus, 6 matches will be returned and each having only one vertex different from the other.

A more immediate concern is how this notion of cardinality truly extends to graphs. In strings, there is an advantage of a strict ordering from left to right, while graphs don't. By just extending the example in Figure 19 with another pattern vertex we see that the specification is ambiguous.

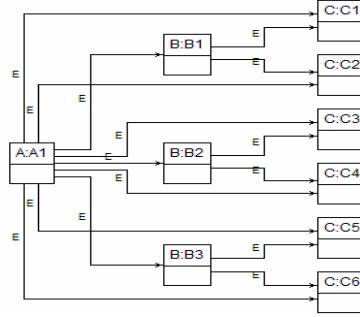
Figure 20(a) shows a pattern having three vertices. There are different semantics that can be associated with the pattern. One possible semantic is to consider each pattern vertex pv to have a set of matches equalling the cardinality of the vertex. Then an edge between two pattern vertices $pv1$ & $pv2$, implies that in a match each $v1, v2$ pair are adjacent, where $v1$ is bound to $pv1$ and $v2$ is bound to $pv2$. This semantic when applied to the pattern in Figure 20(a) gives the graph in Figure 20(b).



(a) Pattern with three vertices



(b) Set semantics



(c) Tree semantics

Figure 20 Pattern with different semantic meanings

The algorithm to search the host graph for a set of matches according to the above-mentioned semantics is given in Appendix B. This algorithm is a direct extension of the algorithm in Appendix A.

The set semantics will always return a match of the structure shown in Figure 20(b), and it does not depend upon the factors like the starting point of the search and how the search is conducted. However, with set semantics it is not obvious how to represent a pattern to match the graph shown in Figure 20(c).

Another possible semantics could be the tree semantics: If a pattern vertex pv_1 with cardinality c_1 is adjacent to pattern vertex pv_2 with cardinality c_2 , then the semantics is, each vertex bound to v_1 will be adjacent to c_2 vertices bound to v_2 . Let $b_1 = (pv_1, V_1)$ and $b_2 = (pv_2, V_2)$ be the bindings for pv_1 and pv_2 respectively.

$$\forall v_1 \in V_1 \exists_{n=1}^{c_2} v_{2n} \in V_2, \wedge e(v_1, v_{2n}) \quad \textbf{Relation 3}$$

This semantics when applied to the pattern gives Figure 20(c). The tree semantic is weak in the sense that it will yield different results for different traversals of the pattern vertices and edges. For the traversal sequence pa, pb, pc the graph shown in Figure 20(c) is obtained while for the traversal sequence pa, pc, pb a different graph as shown in

Figure 21 is obtained. Another problem with tree semantics is that graphs like the one shown in Figure 20(b) cannot be expressed in a concise manner.

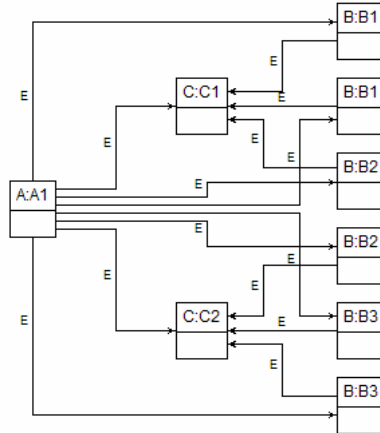


Figure 21 Conflicting match for the tree semantics

Both set and tree semantics discussed so far are incomplete in the sense that certain pattern matches cannot be expressed with them. Choosing either one compromises the expressiveness of the language. Also the tree semantics also brings in a different form of non-determinism because different traversal sequences yield different results. Fortunately, there is a pragmatic solution that solves all the problems: to use a more expressive, extended set notation.

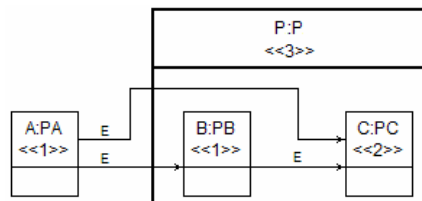
Extending the Set Semantics

If we want to specify a string “sxyxyxy”, we see that “xy” is repeated 3 times. Extending the notation used before we would express it as “s3(xy)”. Using parenthesis we were able to represent the fact that the “xy” sequence should occur 3 times. A similar notion can be used in graphs as well. That is, by grouping vertices of a pattern to form a sub-pattern, a larger pattern can be constructed using these sub-patterns. If a group

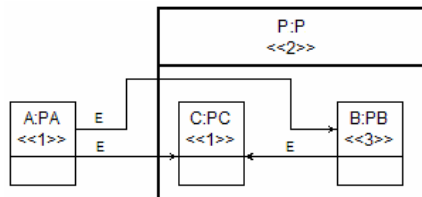
consists of a sub-pattern that has cardinality n then n sub graphs need to be found. Another important point here is that while in strings the ordering of each element of the group is implicit, in graphs we have to explicitly specify the connectivity. Pattern edges that cross groups are used for this purpose.

To illustrate this point, Figure 22(a) shows the pattern that would express the graph in Figure 20(c) and Figure 22(b) shows the graph the expresses the graph in Figure 21. With respect to the pattern P in Figure 22(a) there will be exactly one vertex PB that will connect to exactly 2 vertices of type PC. The larger pattern will consist of the 3 sub patterns of the type described by P. The resulting graph that will be matched is shown in Figure 20(c).

The above exercise illustrates two points. First, set semantics along with the grouping notion can express all the graphs that tree semantics can express and second, the semantics are still precise and map to exactly one graph.



(a) Pattern for Figure 20(c)



(b) Pattern for Figure 21

Figure 22 Hierarchical patterns using set semantics

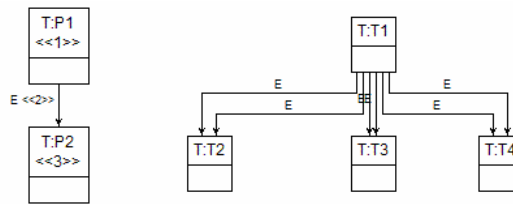
At this point it is apparent that a variety of graphs can be expressed in an intuitive, concise and precise way. However, a large number of graphs are missing from the Grouped Set Semantics (GSS) described above: these graphs are those having more than one edge for the same pair of vertices.

Cardinality for Edges

Adding cardinality to pattern edges helps us express additional graph patterns in a compact manner. Another example is called for and is shown in Figure 23. The figure shows a pattern with cardinality on the edge. The semantic meaning is an extension of Relation 1. Let $b1=(V1,pv1)$ and $b2$ defined as

$$\forall v1 \in V1, v2 \in V2, \exists_{n=1}^C e_n(v1, v2) \quad \text{Relation 4}$$

The extension is that instead of having one edge between each pair of vertices there can be C edges where C is the cardinality of the pattern edge.



(a) Pattern (b) Matching Host graph

Figure 23 Pattern with cardinality on edge.

Variable Cardinality

Sometimes, the sub graph to be found is not of a particular structure but can belong to a family of graphs. Suppose a string needs to be matched such that it starts with

‘s’ and is followed by 1 or more ‘b’s. Therefore, the pattern specification represents a family of strings. This can be expressed with the help of regular expressions, such as “s(b)+”. In the general case the number of ‘b’s can be bound by two numbers, the lower and upper bound. To extend the example let us consider that 5 to 10 ‘b’s could follow the ‘s’. By extending the regular expression notation slightly, we can come up with a notation “s(5..10)(b)”.

Using a similar method for graphs, the notation of cardinality to be variable of the form (x..y), where the lower bound is x and the upper bound is y. Hence a particular pattern vertex should match at least x host graph vertices and not more than y host graph vertices. The upper bound can however be *, representing no limit. This approach can also be used to specify optional components in a pattern by having the cardinality of optional components as (0..1).

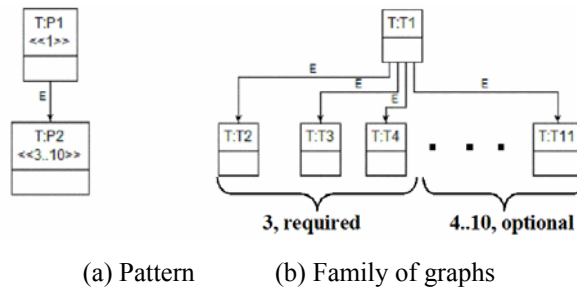


Figure 24 Variable cardinality pattern and family of graphs

Figure 24 presents a variable cardinality pattern. The pattern in Figure 24(a) specifies that 3..10 P2s can be connected to a P1, thus the family of graphs represented is given in Figure 24(b). The required portion must be present while the optional part may or may not be present. Finally the specification language has been extended to express a truly large set of graphs.

However, there are a few problems with variable cardinality. Consider the pattern in Figure 24(a) and suppose that there is a graph having T2..T11 connected to T1 in the host graph. Should the pattern-matching algorithm return only one match namely the entire host graph or all possible sub graphs with cardinality 3, 4 till 10. The way this question is answered is that if more than one match occurs; then both the matches will be returned if and only if neither match is a proper subset of the other. Thus the matches returned would each be maximal and consistent with respect to the pattern.

$$\forall m1, m2 \in M, m1 \not\subset m2 \wedge m2 \not\subset m1 \quad \textbf{Relation 5}$$

Relation 3 states that from the returned set of matches there should not be any two matches such that one is the subset of the other.

This construction yields a precise and consistent language, which can be used to specify complex patterns in a concise manner.

Pattern Graph and Match Definition

After the discussion on the specification of patterns we can now define pattern vertices, edges and graphs.

A pattern vertex PV is a pair: (class, cardinality), where class is a UML class defined in the heterogeneous metamodel and cardinality is a pair (lower bound, upper bound). A pattern edge PE is a 4-tuple (etype, src, dst, cardinality), where etype is the association the edge belongs to. Src and dst are the pattern vertices that the edge is incident upon. The class of these vertices must be identical to the endpoint classes of etype. A pattern graph PG is pair (GPV, GPE), where GPV is a set of vertices in the graph and GPE is the set of edges and $\forall pe = (etype, src, dst, c) \in GPE, src \in GPV \wedge dst \in GPV$.

The definition of a match can also be suitably revised to a pair (MVB, MEB), where MVB is a set of vertex bindings and MEB is a set of edge bindings. Vertex binding is defined as a pair (PV, HV), where PV is a pattern vertex and HV is a set of host graph vertices. Similarly edge binding is a pair (PE, HE), where PE is a pattern edge and HE is a set of host graph edges. The match must satisfy the following properties.

$$\begin{aligned} &\forall EB \in MEB, \text{ where } EB = (pe, HE), pe = (pet, psrc, pdst, cardinality), \\ &\forall he \in HE, he = (het, hsrc, hdst), \exists VBS \in MVB \wedge VBD \in MVB \\ &\therefore VBS = (psrc, hsrc) \wedge VBD = (pdst, hdst) \end{aligned}$$

and

$$\begin{aligned} &\forall EB \in MEB, \text{ where} \\ &EB = (pe, HE), pe = (pet, psrc, pdst, (lower, upper)) \\ &lower \leq C_{HE} \leq upper \\ &\text{and} \\ &\forall VB \in MVB, \text{ where} \\ &VB = (pv, HV), pv = (pclass, (lower, upper)) \\ &lower \leq C_{HV} \leq upper \end{aligned}$$

Graph Rewriting/Transformation Language

The graph transformation language is inspired by many previous efforts such as [69][70][72][80][81]. It defines the basic transformation entity: a production/rule. A production contains a pattern graph. These pattern objects each conform to a type: class or association from the metamodel. Apart from this, each pattern object has another attribute that specifies the role it plays in the transformation. There are three different roles that a pattern object can play. They are:

bind: The object is used to match objects in the graph.

delete: The object is used to match objects, but once the match is computed, the objects are deleted.

new: After the match is computed, new objects are created.

The execution of a rule involves matching every pattern object marked either *bind* or *delete*. If the pattern matcher is successful in finding matches for the pattern, then for each match the pattern objects marked *delete* are deleted and then the objects marked *new* are created. Since the pattern matcher returns all matches for the pattern, there can be a case where a host graph object is deleted from a match while the next match still has a binding for it. The *delete* operation checks for such a situation and if it exists it does not perform the *delete* and returns failure. Thus, only those objects can be deleted that are bound exactly once across all the matches.

Sometimes, the patterns by themselves are not enough to specify the exact graph parts to match and we need other, non-structural constraints on the pattern. For example, “an integer attribute of a particular vertex should be within a range.” These constraints can be described using a constraint language such as Object Constraint Language (OCL) [87], a widely used standard that is directly related to UML, the metamodeling language of GME. There is also a need to provide values to attributes of newly created objects and/or modify attributes of existing objects. This done via “attribute mapping”.

The formal definition of a production is as follows: A production P is a triple (pattern graph, guard, attribute mapping), where

- *Pattern graph* is a graph (in the definitions section).
- *Pattern Role* is a mapping for each pattern vertex/edge to an element of role = {bind, delete, new}.

- *Guard* is a boolean-valued expression that operates on the vertex and edge attributes. If the guard is false, then the production will not execute any operations.
- *Attribute mapping* is a set of assignment statements that specify values for attributes and can use values of other edge and vertex attributes.

Language Realization

The goal of GReAT is (1) to transform models that (a) belong to one meta-model into models that belong to another meta-model or (b) to transform models within one meta-model, and (2) to maintain the consistency of the models with respect to their meta-models. Hence, it is important that the language only allows the user to draw patterns that conform to the meta-models.

To maintain consistency and provide usability in GReAT, the following use case is defined. The use case is supported through the services of the modeling environment (GME).

- The user first imports the input and output metamodels in the form of libraries.
- Next, the user specifies a separate metamodel that defines all the temporary vertices and edges that will be need for the transformation.
- After attaching and specifying these metamodels the user can then draw productions/rules that specify patterns.

Figure 25 shows an example rule. The rule contains a pattern graph, a Guard and an AttributeMapping. Each object in the pattern graph refers to a class in the heterogeneous metamodel. The semantic meaning of the reference is that the pattern object should match with a graph object that is an instance of the class represented by the

metamodel entity. The default action of the pattern objects is *Bind*. The *New* action is denoted by a tick mark on the pattern vertex (see the vertex StateNew in figure). *Delete* is represented using a cross mark (not shown in figure). The *In* and *Out* icons in the figure are used for passing graph objects between rules and will be discussed in detail in the next section.

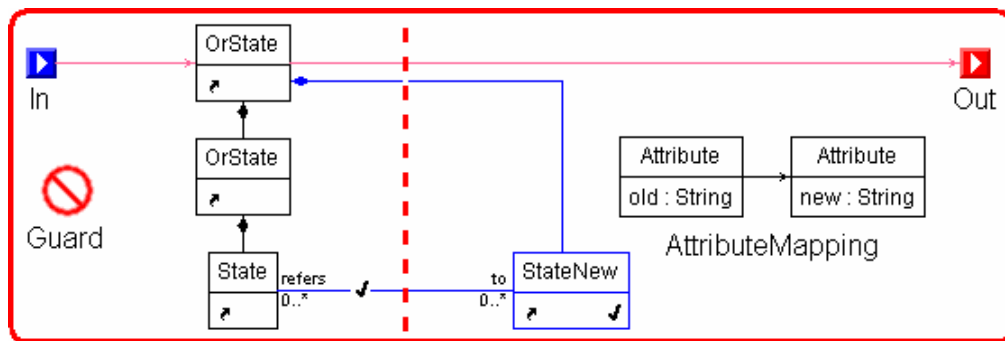


Figure 25 An example rule with patterns, guards and attribute mapping

GReAT relies on UML metamodels for defining patterns. Furthermore, the patterns are also specified in (a superset of the) UML syntax. Since the modeler uses UML for metamodeling it was more intuitive to describe the rules in UML too. By making the user reference each pattern object, the consistency of the patterns and thus the consistency of the transformations is enforced.

The Language For Controlled Graph Rewriting And Transformation

Since the pattern matcher is exponential in the number of pattern vertices there is a need to devise methods to keep this complexity in manageable limits. The performance of the pattern matching can be significantly increased if some of the pattern variables are bound to elements of the host graph *before* the matching algorithm is started (effectively

providing a context for the search). The initial matches are provided to a transformation rule with the help of *ports* that form the input and output interface for each transformation step. Thus, a transformation rule is similar to a function, which is applied to the set of bindings received through the input ports and results in a set of bindings over the output ports. For a transformation to be executed, graph objects must be supplied to each port in the input interface. In Figure 25 the *In* and *Out* icons are input and output ports respectively. Input ports provide the initial match to the pattern matcher while output ports are used to extract graph objects from the rule so that they can be passed along to the next rule. The rules thus operate on *packets*, which are defined as sets of (port, host graph object) pairs.

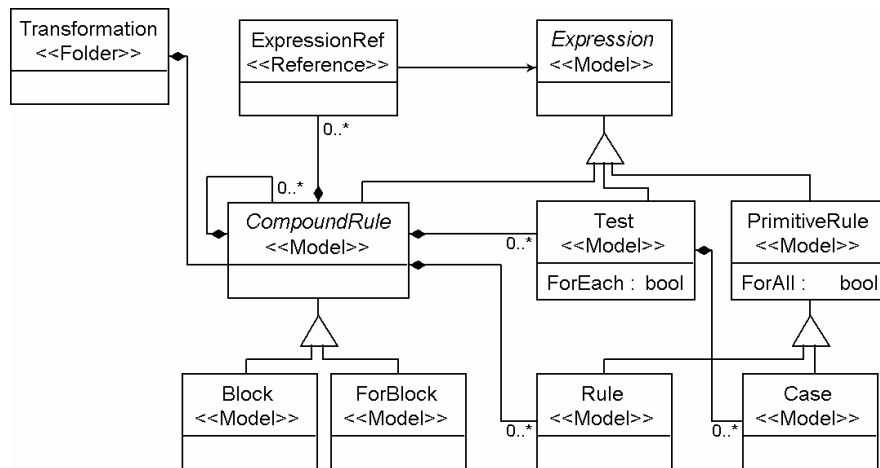


Figure 26: UML class diagram for the abstract syntax classes of GREAT: The core transformation classes

The next concern is the application order of rewriting productions. Classical graph grammars apply any production that is feasible. This technique is good for generating and matching languages but model-to-model transformations often can and need to follow an

algorithm that requires a more strict control over the execution sequence of rules, with the additional benefit of making the implementation more efficient.

In order to better manage complexity in transformation programs it is important to have higher-level constructs, like hierarchical rules and control structures in the graph rewriting language. For these reasons GReAT supports (1) the nesting of rules and (2) control structures. We show these capabilities here using the classes that form the abstract syntax tree of the language. The common abstract base class for the language is *Expression* as shown in Figure 26, and all other constructs like *Rules* and *Blocks* are derived from it. The derivation implies a shared base semantics: all these classes represent some kind of graph transformations.

Figure 27 shows input-output interfaces (*Ports*) of the *Expressions* (*In* and *Out*), as well as sequencing (*Sequence*), the pattern class objects (*PatternClass*) and their connection to the ports (*Binding*). The interface of the expressions allows the outputs of one expression to be the input of another expression, in a dataflow-like manner. This is used to sequence expression execution.

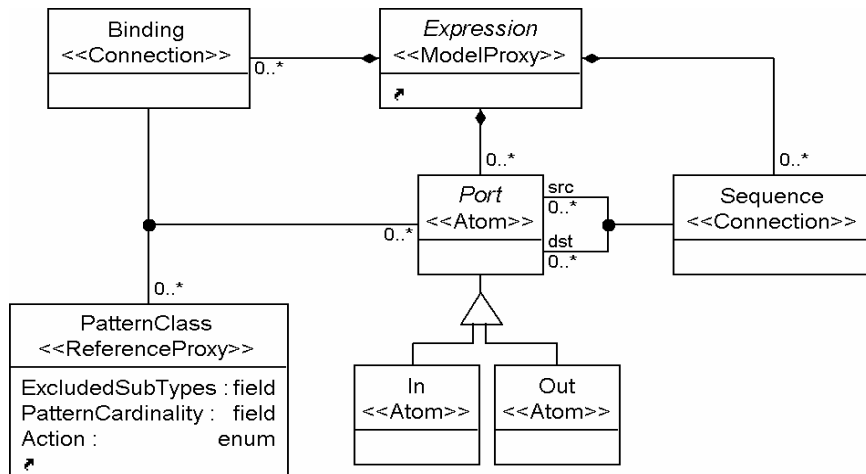


Figure 27: UML class diagram for the abstract syntax classes of GReAT: The interface

A *CompoundRule* can contain other compound rules, *Tests*, and *PrimitiveRules*.

The primitive rules of the language are to express primitive transformations. A *Test* is a special expression and is used to change the control flow during execution.

The control flow language has the following basic control flow concepts.

- Sequencing – rules can be sequenced to fire one after another
- Non-Determinism – rules can be specified to be executed “in parallel”, where the order of firing of the parallel rules is non deterministic.
- Hierarchy – *CompoundRules* can contain other *CompoundRules* or *Expressions*
- Recursion – A high-level rule can call itself.
- Test/Case – A conditional branching construct that can be used to choose between different control flow paths.

Note that the approach followed here can be considered as a highly specialized version of the transformation unit concepts introduced in [95]. The hierarchical rules can be viewed as graph transformation modules, but in GReAT the control condition is restricted. Also, GReAT does not address the issue of transactions, as all rule execution is assumed single-threaded.

Sequencing of Rules

If the output interface of a rule is associated with the input interface of another rule, they will execute sequentially. Figure 28 shows the flow of packets through the rules. The packets are shown as a vertical set of letters where each letter refers to host graph object. The packet objects map to the ports of a rule in the vertical layout. Thus, the top graph object is bound to the top port and so on. Figure 28(a) shows the initial condition where there are two input packets on the input interface of Rule 1. Rule 1 will

fire, consume all its input packets and produce a number of output packets as shown in Figure 28(b). Then rule 2 will fire, consume all its input packets to produce a number of output packets (shown in Figure 28(c)).

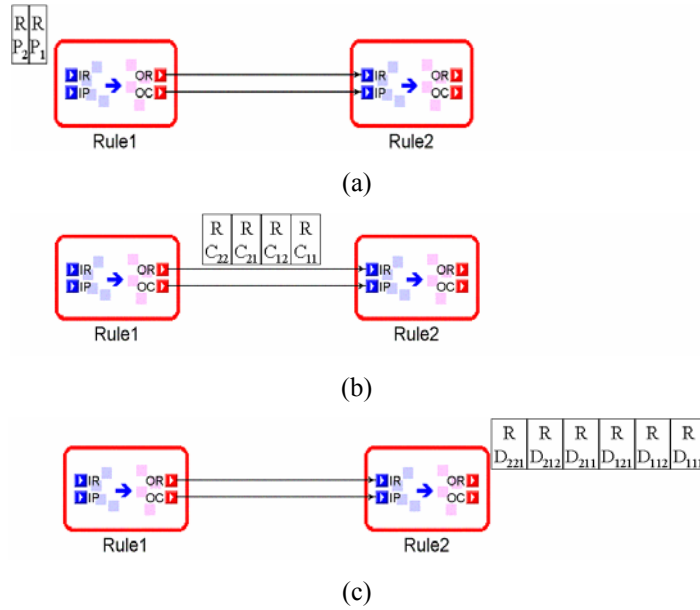
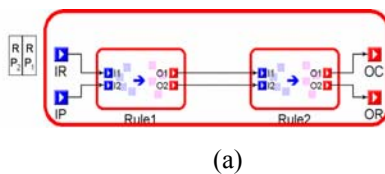


Figure 28 Firing of a sequence of 2 rules

Hierarchical Rules

There are two kinds of hierarchical “container” rules: (1) *Block*, and (2) *ForBlock*. Both *Block* and *ForBlock* have the same semantics with respect to rules connected to them. Thus, if in Figure 28 the rules 1 and 2 were hierarchical, then they would have had the same effects as described above. All the semantic differences are internal to the hierarchical rules.



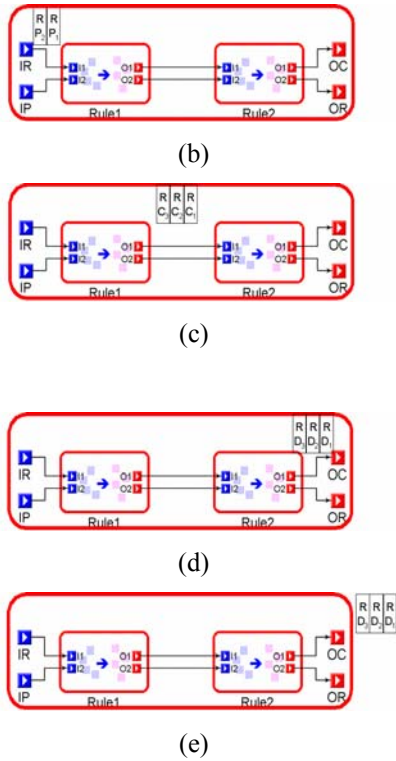


Figure 29 Rule execution of a Block

The *Block* has the following semantics: it will push all its incoming packets through to the first internal rule (i.e. it is same as the regular rule semantics). The input interface of the block can be attached to the input interface of any internal block or to the output interface of the block. In other words the block can send output packets from any internal rule or pass its input packets as output. However, the output interface of a block must be attached to exactly one interface and it cannot be attached to two different interfaces. Figure 29 illustrates the execution of rules within a block. Figure 30 illustrates the case when the output interface of a block is connected to the input interface of the same block.

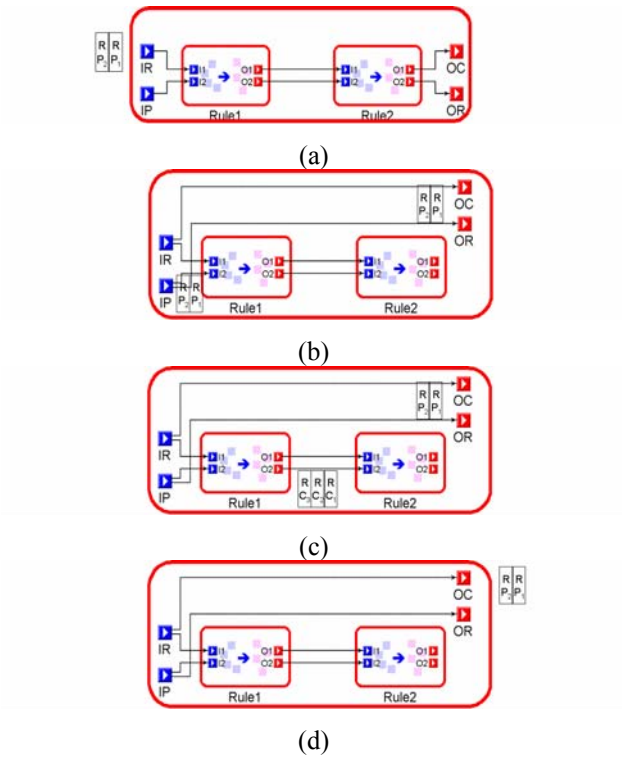
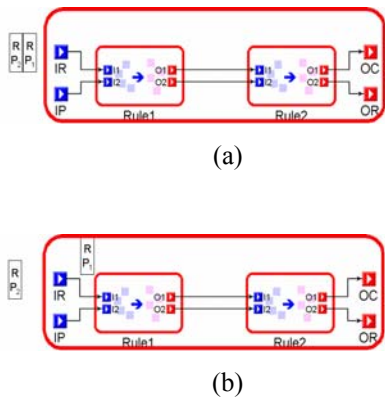
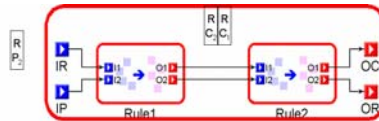


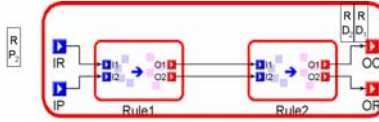
Figure 30 Sequence of execution within a *Block*

The *ForBlock* has different execution semantics than the *Block*. If there are n incoming packets then the first packet will be pushed through all its internal rules to produce output packets and only then the next packet will be taken. The semantics are illustrated with the help of an example in Figure 31.

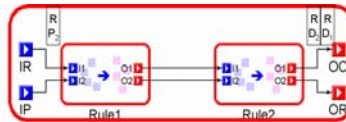




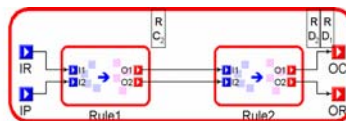
(c)



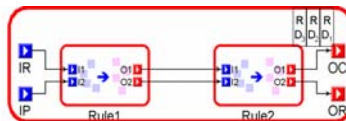
(d)



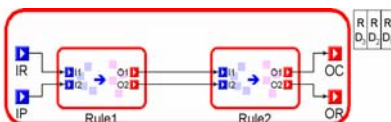
(e)



(f)



(g)



(h)

Figure 31 Rule execution sequence of a *ForBlock*

Similar to the block the input interface of the *ForBlock* can also be associated with the input interface of any internal rule or the output interface of itself.

Branching using test case

There are many scenarios where the transformation to be applied is conditional and a “branching” construct is required. In such cases GReAT supports a branching construct called *Test/Case*.

The external semantics of a *Test/Case* is similar to any other rule. When fired or executed it consumes all its input packets to produce some output packets. The internal working of a test is a bit different from other blocks. In a *Test* all cases get their inputs from the input interface of the *Test*. Unlike a *Block* or a *ForBlock* the execution of the case is not non-deterministic but is based on the physical placement of the cases. The cases are evaluated in a top-down order. Cases can only match, not make changes to the graph. Even if a *Case* succeeds all other cases are executed. This can be considered as a series of *if* statements in a regular programming language without the *else*. There is a construct called *Cut* which if enabled will stop the *Test* after the first successful *Case*.

Figure 32 shows a *Test* with two cases. The *Test* has one input interface and two output interfaces ($\{OR1, OP1\}$ and $\{OR2, OP2\}$). When the test is fired each incoming packet is tested and placed on the corresponding output interface.



Figure 32 Execution of a *Test/Case* construct

The test must contain at least one *Case*, and a case is a rule with no output pattern and no actions. It contains a pattern (containing *bind* objects only), a guard condition and

an input/output interface. If the pattern matches and the guard evaluates to true, then the case succeeds and the input packet given to the case is passed along, otherwise the case fails.

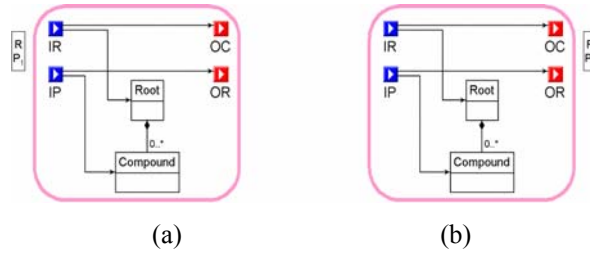
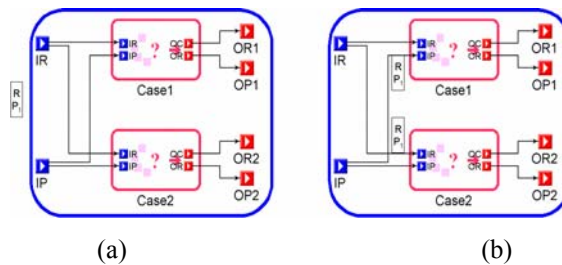


Figure 33: Execution of a single *Case*

Figure 33 shows a case with a successful execution. The input packet has a valid match and so the packet is allowed to go forward. In Figure 34 the execution of a test is shown. An input packet is replicated for each case. Then the input packet is tried with the first case, it succeeds and is copied to the output of the case. Since the *Cut* is not enabled in the first case the packet is tried with the second case, this time it fails and the packet is removed. Finally, after all input packets have been consumed the output interfaces have the respective packets.



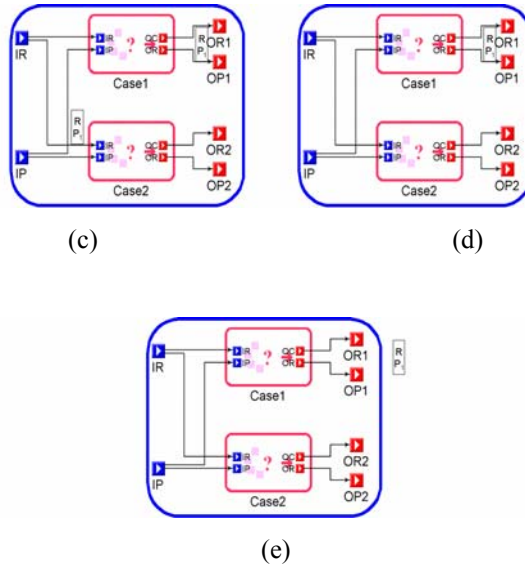
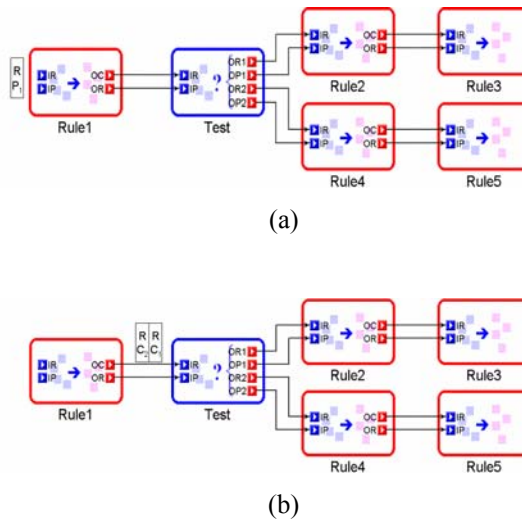
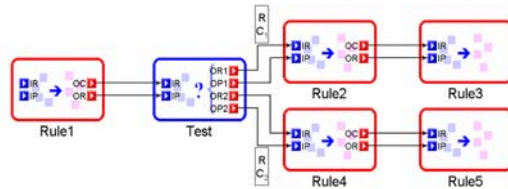


Figure 34 Inside the execution of a *Test*

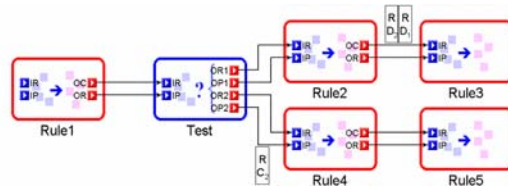
Non-deterministic Execution

When a rule is connected to more than one follow-up rule, or when there is a test with more than one successful case, then the execution becomes non-deterministic. The execution engine chooses a path non-deterministically, and the path that is chosen is executed completely before the next path is chosen.

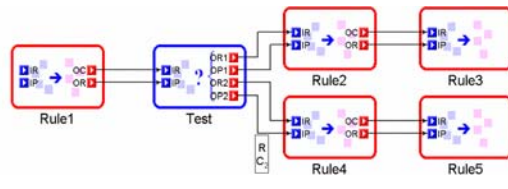




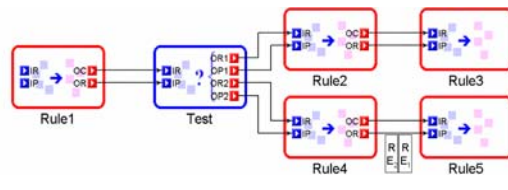
(c)



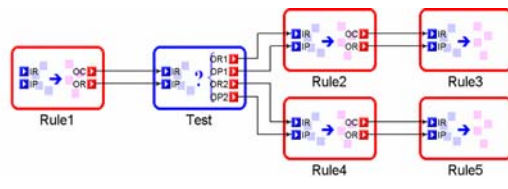
(d)



(e)



(f)



(g)

Figure 35 A non-deterministic execution sequence

Figure 35 shows a non-deterministic execution sequence. Here the non-deterministic execution is caused due to a test/case but it could also have been due to a rule connected to more than one other rule. After the branch there are packets at both the

output interfaces of the test. Thus, both rule 2 and rule 4 are ready to fire. Rule 2 is chosen non-deterministically and fired, followed by the execution of the following rules. This ends at rule 3. Then rule 4 and 5 are fired.

Termination

At one point, the transformation must terminate. A rule sequence is terminated either when a rule has no output interface or when a rule having an output interface does not produce any output packets.

If the firing of a rule produces zero output packets then the rules following it will not be executed. Hence in Figure 35, if rule 4 produced zero output packets then rule 5 would not have been fired.

Enabling Optimized Graph Transformations

This section highlights language features in GReAT that facilitate the development of optimized transformations.

Typed Patterns

It is well known that subgraph isomorphism is an exponential time algorithm in terms of the input graph and the pattern graph $O(n^p)$ where n is the number of nodes in the input graph and p is the number of nodes in the pattern graph. In order to reduce the average case execution time a number of steps can be taken.

The first step is to type the pattern vertices and edges. This restricts the search to a subgraph of the host that only contains the particular types used in the pattern. If we consider a host graph having say T types of vertices and if we assume that the vertices

have even distribution with respect to its type then the time complexity of matching a pattern with P_t types of vertices is $O\left(\frac{P}{T} \times n^p\right)$. Even though the worst case execution time is $O(n^p)$ the expected case execution time will be reduced.

Pivoted Pattern Matching

Another optimization technique is to start the pattern matcher with an initial binding and we have named it “pivoted pattern matching”. In this technique the programmer provides an initial binding for some of the models in the pattern graph to the host graph nodes. The pattern matching is then performed in the context of the initial binding.

In Figure 36, the pattern vertex Pv is initially bound to the host vertex Hv . This restricts the search to the area shown within dotted line. This particular optimization technique works well for sparse graphs. Consider a graph that has an average degree (the number of edges incident on a vertex) of 3 and the greatest distance from the pivot to a vertex in the pattern graph of 2. Then the matching algorithm will only search within a tree of depth 3 starting from the pivoted node. In general the number of host graph vertices included in the search will be c^d where c is the connectivity and d is the depth of the pattern. Hence the order complexity of the matching algorithm is $O(n^p)$ where $n = c^d$ and p is the number of unbound pattern vertices.

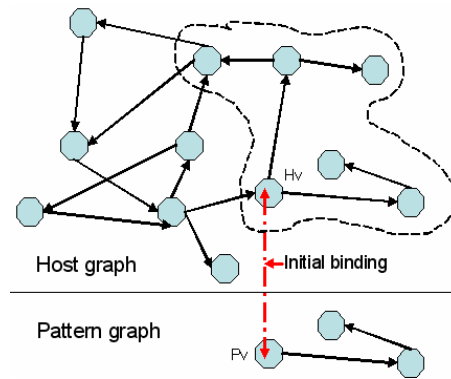


Figure 36 Pivoted Matching

Pivoted pattern matching optimization, when added to the typed pattern vertex technique gives a significant saving because in this case the connectivity of the restricted graph is even less. Figure 37 shows a rule with *In* and *Out* ports that have been used to provide the initial binding. The *OrState* pattern vertex is bound to a host graph vertex supplied by the port labeled *In*.

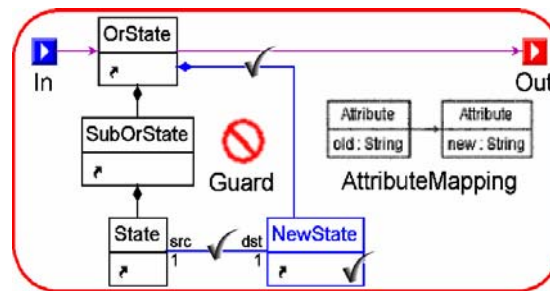


Figure 37 Transformation Rule with pivot

Reusing Previously Matched Objects

The next optimization technique used in the GReAT is the called “Reusing previously matched objects”. The idea here is to cache previously found results and pass it on to subsequent rules as the initial binding.

For example, in Figure 38, there are two rules. The first rule gets an input binding for *Parent* and finds all *ChildA*, *ChildB*, *Assoc* triples that correspond to the pattern. In the subsequent rule these triples are required to perform an action. Instead of finding the pattern again, the first rule passes the triples along to the next rule. For the next rule they serve as the initial binding. When a rule executes it can produce multiple matches. Each match produces a host graph object for each output port and this coherent set of objects is called a packet. These packets are sent to the subsequent rules as one unit.

User Controlled Traversal

GReAT supports hierarchical specification of transformation rules. High-level rules can be created by composing a sequence of primitive rules. There are two kinds of high-level rules in GReAT: *Block* and *ForBlock*. The execution semantics of the *Block* is to pass all input packets to the first contained rule, the outputs packets created by it are passed to subsequent rules and so on. After all packets have been processed and all output packets of the *Block* have been generated, the *Block* returns control to its parent. Semantics for the *ForBlock* is to pass one input packet at a time through all the contained rules. After the first packet has been processed all the way to the output of the *ForBlock* the next packet is processed. These two constructs enable the user to choose different traversal strategies. A *Test/Case* is also available in GReAT. It can be used to choose between different execution paths, during the transformation and is similar to the ‘if’ statements in programming languages.

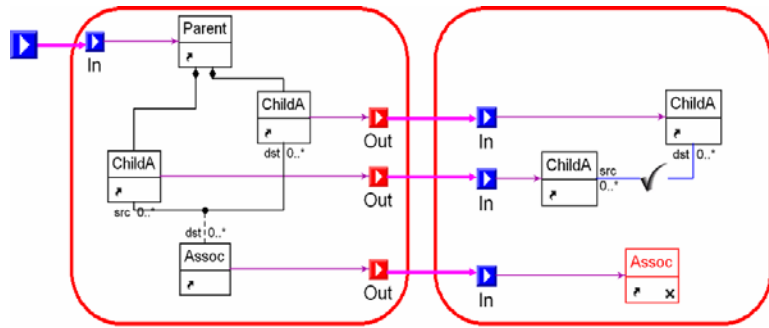


Figure 38 Sequence of rules with passing of previous results

CHAPTER V

THE EXECUTION FRAMEWORK FOR GREAT

This Chapter describes the execution framework that we built for GReAT. This infrastructure can be divided into the following parts.

1. **Concrete Syntax:** the realization of the transformation language.
2. **Abstract Syntax:** a syntax that is void of any concrete representation such that various concrete representations can be mapped to this format.
3. **Execution Engine:** a virtual machine that can execute GReAT programs on a given input to produce output.
4. **Debugger:** debugging support on top of the virtual machine to provide debugging functions such as single step as well as a visual debugging interface.
5. **Code Generator:** the equivalent of a compiler that will consume a GReAT program and produce C++ code that has the same behavior as the GReAT program.
6. **IDE, Integrate development environment:** an environment that consists of an editor, concrete syntax, mapping to abstract syntax and integration with the engine, debugger and code generator.

These components of GReAT will be discussed in details in the following sub sections.

Concrete Syntax

The concrete syntax of GReAT is implemented by a paradigm called UML Model Transformer (UMT) and it contains three parts.

- (1) A metamodeling syntax that allows users to attach metamodels in the form of UML class diagrams and to create temporary/cross associations.
- (2) A concrete syntax for the transformation. This includes syntax for pattern specification, transformation specification and control flow specification.
- (3) Syntax for configuring the execution of the various transformations.

The metamodeling syntax is a restricted subset of UML class diagrams. Entities in this language are Package, Class, Association, Association Class, Composition, Inheritance and OCL constraints. These entities have the same semantics as in the UML specification. In UMT any number of UML packages can be attached. Typically, in a transformation one package is attached for the input domain, another for the output domain and a third for temporary vertices and links. In general, there are no restrictions on the number of domain that can be used.

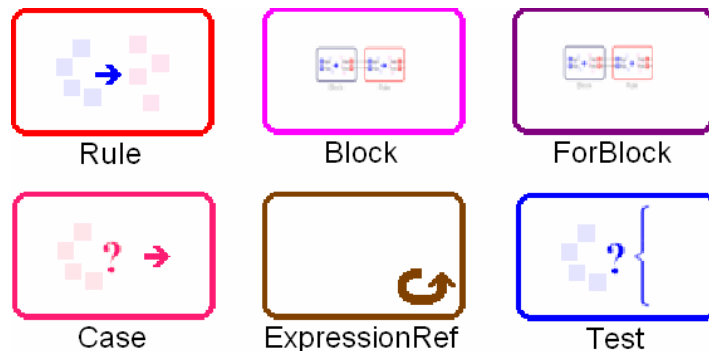


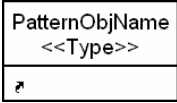

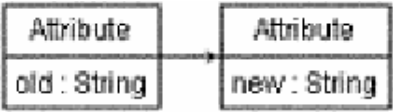


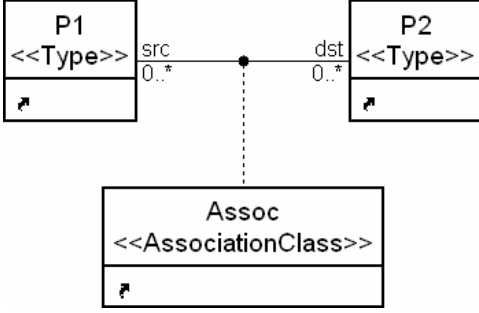


Figure 39 Concrete syntax of the different expressions in GReAT

The abstract syntax of the transformations, expression and their interfaces has been shown in Figure 26 and Figure 27. The concrete realizations have been shown in Figure 39 and Table 3. The figure shows the concrete syntax for the primitive expressions: Rule and Case, compound expressions: Block, ForBlock, Test and the ExpressionRef. Table 3 shows the concrete syntax of the expression interfaces and pattern graphs. It also shows the attributes each entity has.

Table 3 Concrete Syntax of the pattern graph and the rule interface

Entity Kind	Concrete Syntax	Attributes
In		
Out		
Pattern Class		Action PatternCardinality Reference
Guard		ExpressionString
Attribute Mapping		ExpressionString
Pattern Association		Action PatternCardinality

Pattern Composition	 <pre> classDiagram class P1["P1 <<Type>>"] class P2["P2 <<Type>>"] P1 "1" *-- "*" P2 </pre>	Action
Pattern Association with association class	 <pre> classDiagram class P1["P1 <<Type>>"] class P2["P2 <<Type>>"] class Assoc["Assoc <<AssociationClass>>"] P1 "0..*" -- "0..*" P2 : src, dst Assoc .. > P1 Assoc .. > P2 </pre>	

All these concrete syntactic elements come together to form the UMT language. Instances of the language can be seen in Figure 25, Figure 28, Figure 29, Figure 30, Figure 31, Figure 32 and Figure 33.

The concrete syntax of for capturing the configuration information is available in Appendix E.

Abstract Syntax

Three abstract syntax formats have been developed to capture different types of information required for the execution of the transformation tools. These are (1) the Graph Rewriting (GR) format to store the transformation rules, (2) an xml based format to store UML metamodels and (3) GReATConfig, another XML based format to store the configuration information.

The GR format has been described using UML class diagrams and has an XML schema defined for it. The UML class diagram of the GR format (Figure 40) shows the data structure representation. The abstract base class *RuleBase* is the basic element of the

transformation. It can be realized either as a *RewritingRule* or as a *RuleProxy* to a rule, where a rule proxy is a reference to another rule. The *RewritingRule* has an attribute called *ruleType* and this field can be one of the following {Rule, Case, Block, ForBlock, Test, ForTest}. This attribute was used to capture the different rule information in order to keep the language flexible such that new rules types could be easily added. The control flow information is captured using the sequence association between rules.

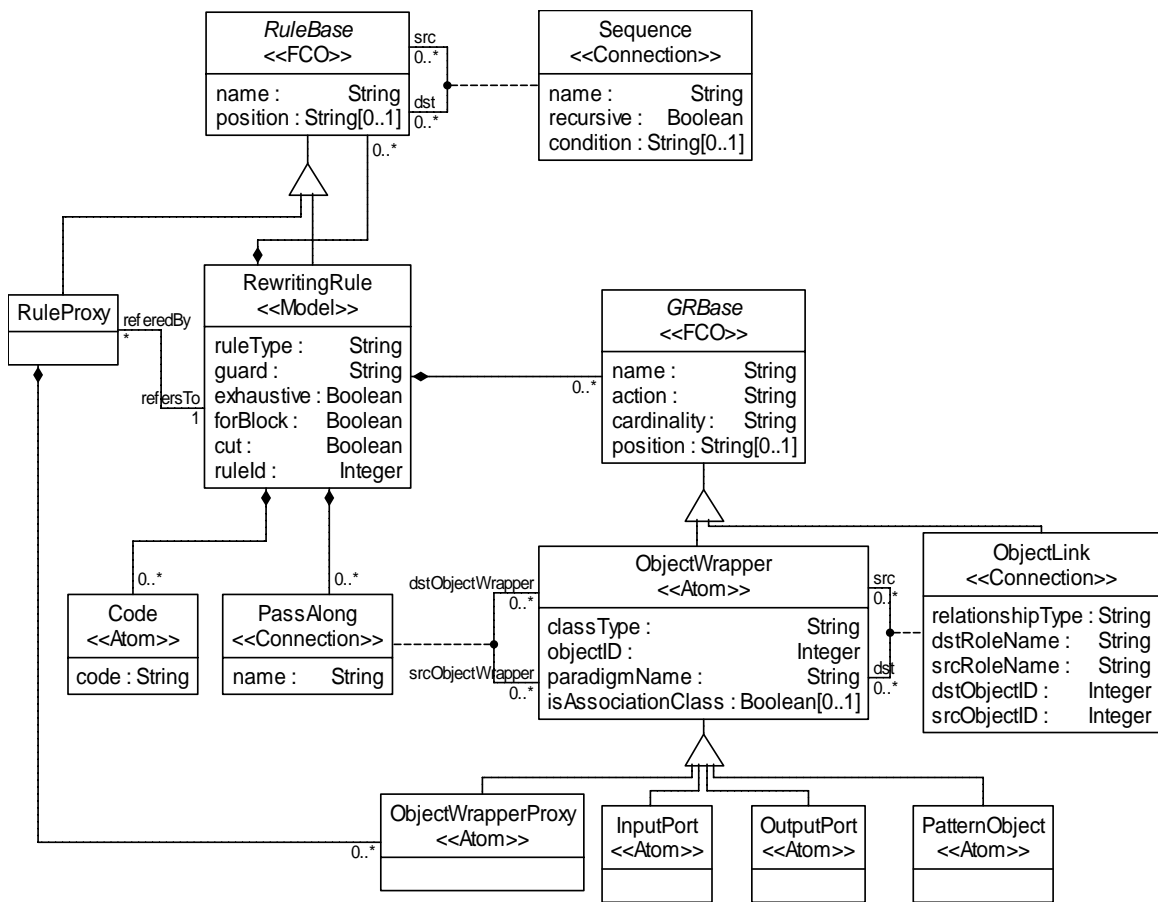


Figure 40 GR: the abstract syntax of GReAT

Rules can contain *GRBase* the base class for pattern objects, pattern links and input/output ports. Primitive rules only contain pattern objects and links, while compound

rules only contain input and output ports. Each pattern object and link has a attribute called action which states the role the object plays in the pattern. The roles as we know can be either CreateNew, Bind or Delete.

Rule proxies refer to other concrete rules. This is used to make a call to a previously defined rule. Rule proxies can only contain *ObjectWrapperProxies*. These are references to the interface of the original rule and are used to capture the data relations between the proxy and preceding, succeeding rules.

The GR format depicting the abstract syntax helps to decouple the implementation of the language from its concrete syntax. There could be a different concrete syntax that can be used for the specification of transformations. Transformations specified in a particular concrete syntax can then be mapped to the GR format for execution.

The GReAT Config format has also been defined using a UML class diagram and an automatically generated xsd. (see Appendix E)

Execution Engine

The realization of a language can be achieved using various methods. The first is by creating an interpreter, a program that will read, understand and execute language at runtime. The interpreter can be regarded as a virtual machine that can execute sentences of a language. The other approach is that of compilation where the sentences of the language are translated to assembly or machine code. The machine code can then be directly executed. In languages such as Java, the classical separation of interpreters and compilers do not hold true. In Java, a compiler is used to convert Java programs to a byte

code format and then an interpreter called the Java Virtual Machine is used to execute the byte code.

GReAT uses a similar approach in the execution engine. The concrete syntax of GReAT is first compiled into the abstract syntax representation (the GR format). A virtual machine called the Graph Rewriting Engine (GRE) has been developed that can execute transformations represented in the GR format.

The input and output of GRE are typed attributed graphs that conform to a domain specification. This adds another level of complexity where the data representation has to be discovered at runtime. Due to this reason handling of input and output graphs is also complicated.

The primary modules of the GRE are:

1. **Metamodel independent data management layer.** This part is required by the GRE to abstract out data access such that the traversal, modification and creation of graphs can be dealt in a uniform manner by the transformation engine. This layer uses the graph and its metamodel to identify and interact with the graphs.
2. **Transformation traversal layer.** This part of GRE is responsible for reading and understanding the transformation specification. It is also known as the sequencer (see Figure 41). It starts from the start rule and is responsible for calling the rule executor with the correct inputs and passing the outputs of the executor to the next schedulable rule.
3. **Sequencer.** This can be considered as the scheduler of GRE. It decides the order of execution of the rules based on the rule type and data

availability. It is also responsible for making the packets available for the rule and for passing the packets to the next rule after execution.

4. **Rule Executer.** Once a rule has been selected to fire, it consumes each input packet one at a time to perform pattern matching and executing the effector. The pattern matching is the core of the transformation engine and has been implemented based on the algorithms described in the pattern specification language section of Chapter IV.

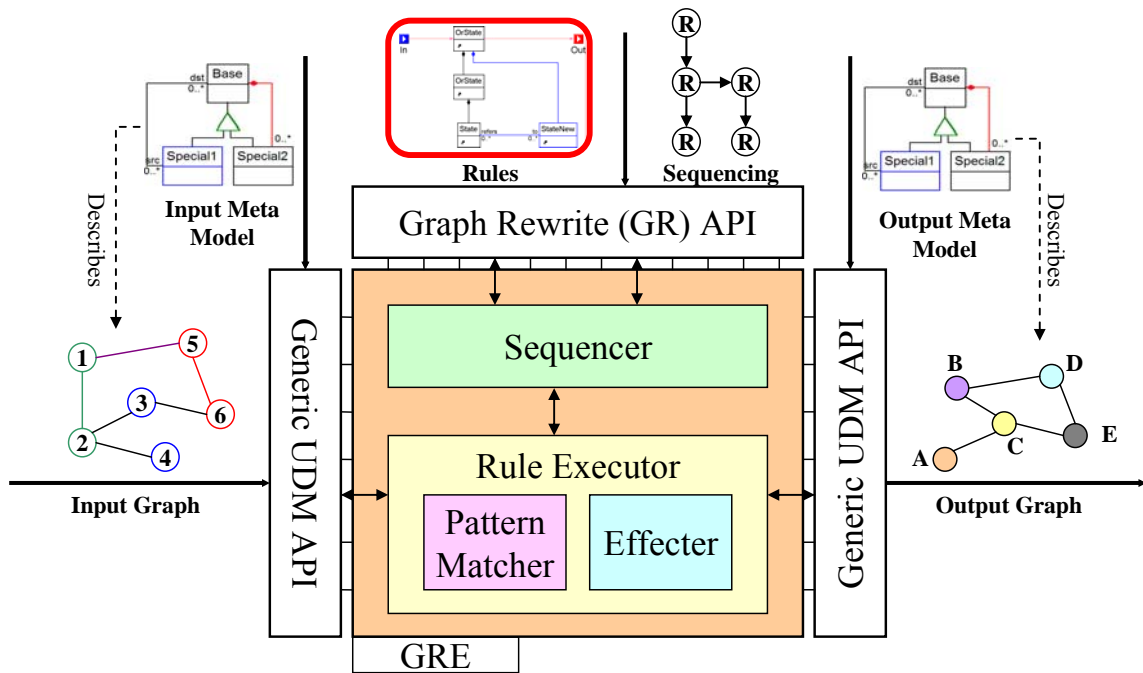


Figure 41 High-level block diagram of GRE

Figure 41 shows the high-level block diagram with the essential features of GRE. The input and output graphs, along with their metamodel are accessed through UDM's [88][89] generic API. UDM abstracts out the storage format for the input and output and

provides uniform, metamodel independent access to the models. The GR specific API is used to access the transformation specification since the GR format will seldom change.

The internals of GRE consist of the sequencer and rule executor. The sequencer is implemented as a hierarchical stack machine. Within a parent rule the first step is to add all the ready-to-fire rules to the stack. The next rule to be fired is fetched from the stack. The differences in the rule types are maintained using different subroutines that add and remove elements from the stack in different ways. The execution semantics of these compound rules has been described in detail in Figure 30.

```

Function Name: ExecuteBlock
Inputs      :    1. List of Packets inputs
                2. Expression block
Outputs    :    1. List of Packets outputs
outputs = ExecuteBlock(block, inputs)
{
    Stack of Rules ready_rules
    foreach next_rule of block.next_rules()
    {
        if(next_rule equals block)
        {
            outputs.Add(inputs )
        }
        else
        {
            ready_rule.Push(next_rule, inputs)
        }
    }
    while( ready_rules.NotEmpty())
    {
        current, arguments = ready_rules.Pop()
        return_arguments = Execute(current, arguments)
        For Each next_rule of current.next_rules()
        {
            if(next_rule equals block)
            {
                outputs.add(inputs)
            }
            else
            {
                ready_rule.Push(next_rule, inputs)
            }
        }
    }
    return outputs
}

```

Figure 42 Block execution algorithm

The execution of the *Block* (see Figure 42) consists of a *ready_rules* that is initialized with the rules that are connected to the input interface of the *Block*. The stack machine then runs till there are no more rules in the stack. The top of the stack is

popped and executed, then the rules that are connected to the output interface of the executed rule are placed onto the stack. When a rule is fired, all incoming packets are passed to it. The execution of the *ForBlock* is slightly different as described in Figure 31. In the *ForBlock* the entire rule chain is executed with one packet at a time. This is achieved (see Figure 43) by calling the *Block* execution for each input packet to the *ForBlock* and then gathering the output packets.

```

Function Name: ExecuteForBlock
Inputs      :    1. List of Packects inputs
                2. Expression forblock
Outputs    :    1. List of Packects outputs
outputs = ExecuteForBlock(forblock, inputs)
{
    List of Packects outputs
    foreach input in inputs
    {
        returns = ExecuteBlock(forblock, input)
        outputs.Add(returns)
    }
    return outputs
}

```

Figure 43 For block execution algorithm

The *Test* is similar to a set of “if” statements without the “else” part. Since the default semantics are that an input packet will be tested with all the cases and more than one case may succeed, there is a requirement for an exclusive style of branching so that only one case succeeds. A variant of this behavior is achieved using a special attribute of a *Case* called the “cut”. When *Case* has its “cut” behavior enabled, if the case succeeds on a given input, the input will not be tried with the subsequent cases. If each case in a test has the “Cut” enabled, then the test will behave like an if-elseif-else programming construct. To implement the “cut” an explicit ordering of the cases is required. The order of testing cases is derived from the physical placement of the case within the test, in the graphical model. The cases are evaluated from top to bottom. If there is a tie in the y co-

ordinate then the x co-ordinate is used from left to right. Figure 44 shows the execution algorithm of the *Test*.

```

Function Name: ExecuteTest
Inputs      : 1. List of Packets inputs
                2. Expression test
Outputs    : 1. List of Packets outputs
outputs = ExecuteTest(test, inputs)
{
  List of Packets outputs
  List of Cases cases =
    test.cases_in_sequence()
  for each input in inputs {
    for each case in cases {
      returns = ExecuteCase(case, input)
      outputs.Add(returns)
      if(case has a cut and return exist)
        break
    }
  }
  return outputs
}

```

Figure 44 Test execution algorithm

Once a primitive rule is selected for execution the rule executor takes control. There are primarily two functions of the rule executor. The first is the pattern matcher and the second is the effector. Figure 45 describes the algorithm executing a production (a “rule”). This algorithm calls the pattern matcher described in Appendix A and B. A “Packet” provides the initial binding required by the pattern matcher and the “Effector” function performs deletion and creation of objects. All the vertices/edges marked for deletion are deleted and vertices/edges marked for creation are created. After all the structural changes have been made, the attribute mapping specification of the rule is executed on the match to changes the attribute values.

```

Function Name : ExecuteRule
Inputs       : 1. Rule rule (rule to execute)
                2. List of Packets inputs
Outputs     : 1. List of Packets outputs
outputs = ExecuteRule(rule, inputs)
{
  List of Packets matches
  List of Packets outputs
  for each input in inputs

```

```

{ matches = PatternMatcher(rule, input)
  for each match in matches
  { if match doesn't satisfy guard
    matches.Remove(match)
  }
  for each match in matches
  { Effector(rule, match)
    outputs.Add(match)
  }
}
return outputs
}

```

Figure 45 Algorithm for rule execution

Graph Rewriting Debugger (GRD)

The success of a programming language often depends on the quality of the error messages a compiler provides and usefulness of the debugger to find and fix semantic errors. With this in mind a debugger for GREAT was developed. Graph Rewriting Debugger (GRD) consists of the following parts.

1. An extension to GRE to allow the transformation to break and single step.
2. A command line debugging interface that allows users to set break points, single step and retrieve stack information.
3. A front end GUI that allows the same features in an interactive environment where the transformation, transformation call stack and input/output packets can be visualized.

The GRD was not developed by me and has been mentioned here for the sake of completeness.

Code Generator

For the sake of efficiency the transformations should have a compiler that converts the transformation specification into code. In the case of GReAT the compiler is composed of two stages, (1) the front-end that converts the concrete syntax to GR and (2) the back-end code generator that generates C++ code from the GR format.

If we write the Graph Rewriting Engine (GRE) of GReAT as a function it will have the following signature:

$GRE : (I \times M_I \times M_O \times T) \rightarrow O$, where

- M_I, M_O - metamodels. A Metamodel is a graph that defines the graph grammar of the input/output models.
- I - input model. A graph that conforms to the metamodel M_I .
- O - output model. A graph that conforms to the metamodel M_O .
- T - transformation. Is a graph rewrite/transformation specification [85].

The Code Generator performs a partial evaluation of the GRE function to produce code specific to a given transformation and input/output metamodels.

$CG : (M_I \times M_O \times T) \rightarrow (T_C : I \rightarrow O)$

The justification for the partial evaluation is that the transformation and the metamodels make up the invariant part of transformation system. The same transformation is typically run on multiple inputs over a course of time. We argue that once the transformation and the modeling paradigm(s) reach a mature state, the transformation should be compiled into a high-performance executable that is capable of performing transformations in an efficient way.

By treating the metamodels as invariants, the CG can generate code that manipulate input and output models using paradigm-specific API's. These API's are generated by Universal Data Model (UDM), a framework that provides object-oriented C++ interfaces to programmatically access input/output models. UDM can generate a domain specific custom API with type-safe access methods for object creation/removal, link creation/removal, and attribute setters/getters [89]. The transformation executable can be built by compiling the generated transformation files and the paradigm-specific API files [85].

Comparison of CG with GRE

In this section a comparison of the execution time of the GRE and the Code Generator is presented. Two transformation problems have been chosen for the comparison. These transformations are:

1. $Df \rightarrow Fdf$: Transform Hierarchical dataflow to its equivalent Flat dataflow representation.
2. $Hsm \rightarrow Fsm$: Transform Hierarchical Concurrent State Machine (HCSM) to its equivalent Finite State Machine (FSM).

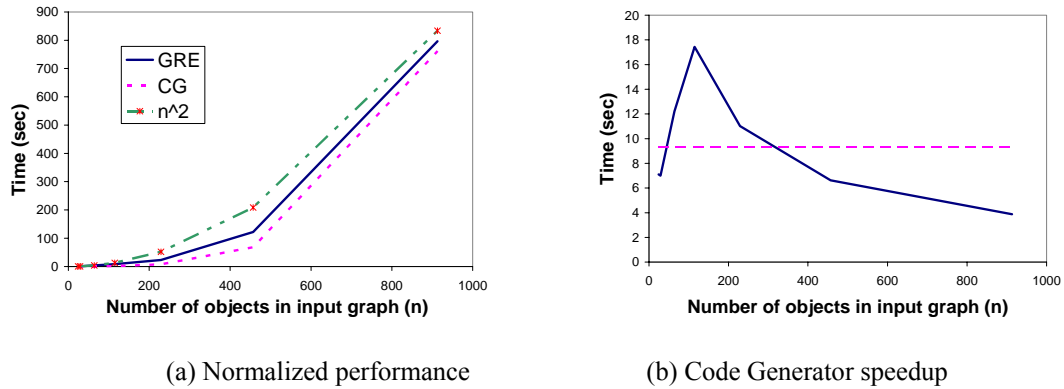
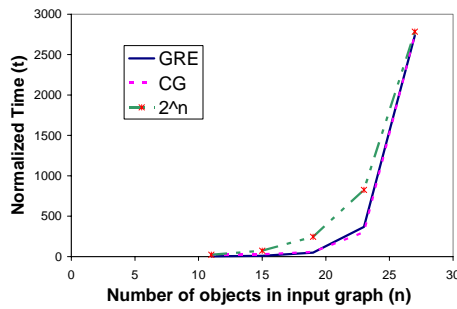
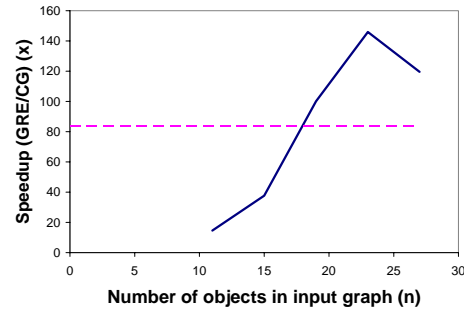


Figure 46 Performance graphs for Df→Fdf

To evaluate the performance of CG in comparison with GRE, the Df→Fdf transformation was executed on 7 different input graphs. The size of these graphs varied from 24 vertices to 914 vertices. Execution times of GRE and CG were measured for all the inputs. Figure 46 (a) is a plot of the input graph size (n) vs. normalized execution time for both GRE and CG. Matlab's polyfit function was used to find the closest fitting polynomial or exponential for the results and the second order polynomial yielded the best results. For this reason the n^2 plot is also shown in Figure 46. From the graph one can see that the order complexity of the transformation doesn't change significantly between GRE and CG and is governed by the complexity of the transformation algorithm. Experimentally it has been seen that the transformation algorithm's complexity is approximately $O(n^2)$. Figure 46 (b) shows the graph of n vs. *speedup* achieved by the code generator. The dashed line in the graph represents the average speedup of 9.3x. From the graph it is observed that the speedup varies within a bound ranging from 4x to 18x.



(a) Normalized performance



(b) Code Generator speedup

Figure 47 Performance graphs for $Hsm \rightarrow Fsm$

For $Hsm \rightarrow Fsm$, 4 input graphs were used. These graphs only had parallel states and varied from 11 vertices to 27 vertices. Execution times of GRE and CG were measured for all the inputs. Figure 47(a) is a plot of the input graph size (n) vs. normalized execution time for both GRE and CG. The polyfit function was again used and this time an exponential to the base 10 yielded the closest results. For this reason Figure 47 also shows the 2^n plot. From the graph we can see that the order complexity of the transformation doesn't change between GRE and CG and is governed by the complexity of the transformation algorithm. Figure 47(b) shows the graph of n vs. *speedup* achieved by the code generator. The dashed line in the graph represents the average speedup of 83.3x. From the graph we can see that the speedup varies within a bound ranging from 14x to 119x. The 14x speedup was observed for very small models and could be because of a constant runtime overhead. A speedup of $\sim 100x$ was observed consistently for larger models.

From the experiments we see that the user is able to specify transformations with polynomial characteristics. This can be attributed to the language features provided in

GReAT. On the other hand exponential algorithms can also be specified as in the case of $Hsm \rightarrow Fsm$.

The second conclusion is that the order complexity of the transformation remains the same for both GRE and CG. This is an expected result because the code generator does not perform any modifications that can provide a gain in order complexity.

The speedup does not seem to have a definitive trend with respect to the input size but varies a lot from one kind of transformation to another. $Df \rightarrow Fdf$, an $O(n^2)$ transformation yielded an average speedup of $\sim 9x$ while the $Hsm \rightarrow Fsm$, an $O(2^n)$ transformation yielded an average speedup of $\sim 85x$. These results make us believe that the speedup is dependent on the complexity of the transformation. For higher complexity transformations the speedup appears higher.

One possible reason for such a result can be based on the percentage of the total execution time spent in *pattern matching* as opposed to *packet passing* and other housekeeping work. Since a higher order complexity algorithm will spend more time in the *pattern matcher*, and the code generator partially evaluates the pattern matcher, a better speedup is observed. When the time complexity of the algorithm is small and the size of the models is large the packet-passing/housekeeping overhead is a large percentage of the total execution time and the speedup observed is less [85].

Integrated Development Environment

The integrated development environment consists of an editor that is aware of the concrete syntax of the transformation language, integration of the interpreter, debugger and compiler within the same environment.

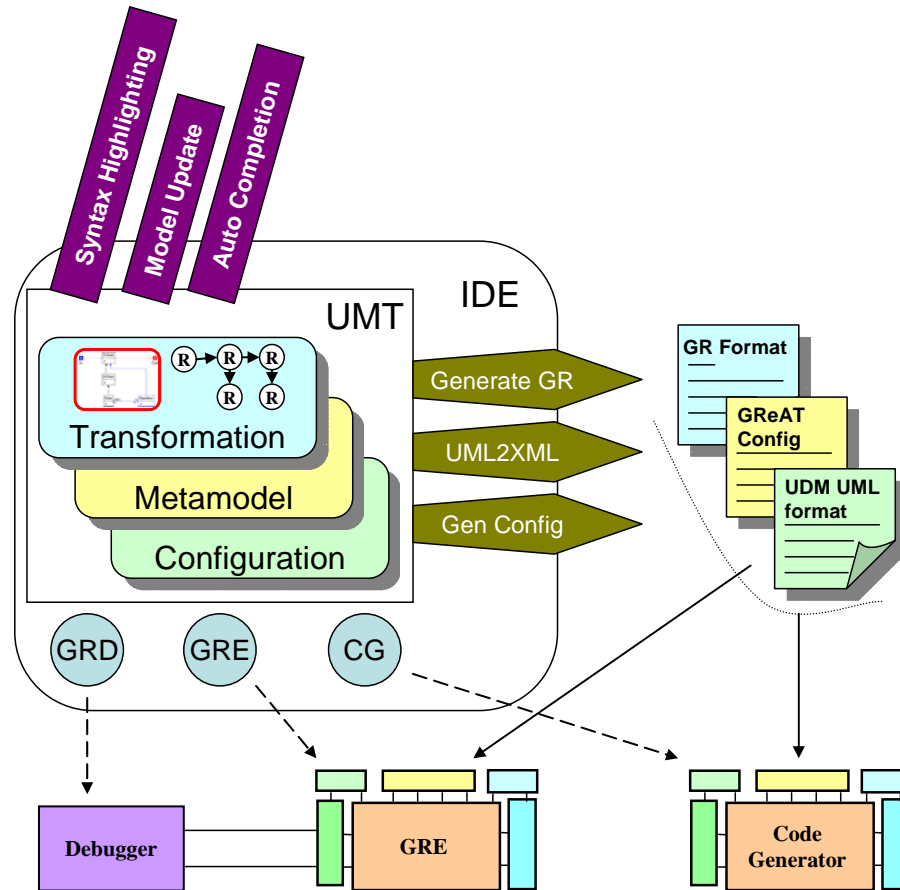


Figure 48 Block diagram of the GReAT IDE

The GReAT IDE is centered on the concrete syntax called UML Model Transformer (UMT). A metamodel for UMT was created to configure GME as a UMT editor. Around this editor other features were added to convert the editor to become an IDE. The suite of tools developed around UMT can be classified into three categories. (1) Model development tools, (2) Model transformation tools and (3) Execution invocation tools. Model development tools are those that are used assist the process of model building. The model transformation tools are used to convert the concrete syntax into the intermediate representations. Execution invocation tools are basically a set of GUI based

access points to the GRE, GRD and CG. They help the user to run, debug and test the transformations within the same framework.

Model Development Tools

To facilitate the development of transformations using the concrete syntax the model editor should make the model development process as user friendly as possible. GME provides domain-specific editing interface where only syntactically correct models can be created. However, it would be better to prevent semantic errors or, at least, report them early. Static semantic errors can be reported with the help of OCL constraints that are evaluated during the model building process. Some errors can be caught and automatically corrected.

Keeping this in mind a set of tools was developed for three primary reasons:

1. **Automatic update of metamodels:** Metamodels are developed and modified in their own files and need to be kept in sync with the metamodels in the transformation. For this reason a metamodel update tool was created that would help copy the modified metamodel into the transformation and update all references from the old one to the new.
2. **Syntax highlighting:** In the transformations pattern objects can have one of three possible actions, namely, Bind, CreateNew and Delete. Each action is associated a different color and different visual representation to clearly distinguish them.
3. **Automatic inference of pattern attributes:** Some combinations of attributes are invalid and are automatically corrected. For example, if a pattern object is marked CreateNew, then its composition with the

parent can only have one action and that is CreateNew. Similarly, there are many cases where the values can be automatically filled. Another example is for the role names on the association. When a pattern association is created, based on the typed of the source and destination object the roles on both ends of the association can be identified from the metamodel.

Execution Invocation Tools

Apart from supporting the development of the transformations, the IDE should also support the execution in an effortless manner such that rapid evaluation of the transformations is supported. For this reason the three execution tools: GRE, GRD and CG have been integrated with the GReAT IDE. This has been achieved by providing a GUI front-end to these that that can be invoked from within the framework.

CHAPTER VI

A CASE STUDY – SIMULINK/STATEFLOW TO HSIF

In this Chapter the solution for a challenge problem is described to demonstrate the use of GReAT. The challenge problem chosen for this task is the semantic translation from Matlab Simulink/Stateflow (MSS) to Hybrid System Interchange format (HSIF) and it can be posed as follows: Given the model of a dynamic system in MSS, compute an equivalent dynamic system model in HSIF which produces the same execution traces when executed, given the operational semantics of HSIF. For pragmatic reasons this constraint was relaxed. First, MSS includes procedural components which are impossible to express in HSIF. To overcome this, restrictions were imposed on MSS that only a subset of the MSS modeling language would be translated. Second, HSIF was defined using mathematical definitions in English, and not operationally (i.e. not via a simulation algorithm). Therefore, a mapping between constructs available in HSIF (e.g. discrete locations, differential equations, transition guards, etc.) and similar constructs in MSS had to be designed such that the two models described the same dynamic system.

In the subsequent sections we describe the inputs and the outputs of the transformation, specify the translation strategy, describe how the transformation was specified in GReAT, give an illustrative example for the use of the translator, and describe the user experience in using GReAT.

The Inputs and Outputs of the Semantic Translator

The output: HSIF

HSIF is an interchange format that allows representation of hybrid systems using dynamic networks of hybrid automata. The detailed specification is available in [98]. The automata in HSIF follows the definition of hybrid automata (HA) [99] with a finite number of locations (or discrete states), where each location has a number of differential and algebraic equations associated with it. Differential equations capture continuous time dynamics in that location, while algebraic equations describe dependencies among variables. HSIF is capable of expressing networks of hybrid automata, where the automata can interact with each other using signals and shared variables. Signals are single writer-multiple reader variables that follow synchronous semantics, while shared variables can have multiple writers and multiple readers.

The input: A subset of the MSS language

Simulink has a rich set of model elements (Simulink blocks) covering various areas of signal processing. In Simulink continuous dynamics and discrete behavior can be mixed arbitrarily. On the other hand, HSIF has a clean separation between continuous and discrete behavior. Mapping arbitrary MSS models with complex interactions between continuous and dynamic behavior to HA is a difficult problem. The solution was to choose a subset of Simulink/Stateflow that maintains a clean separation between the continuous and discrete behavior. A subset of the primitive blocks from MSS was carefully chosen such that it provides a useful coverage. The supported Simulink blocks are as follows:

- **Continuous time blocks:** Integrator, State-space, Transfer Function, Zero-Pole
- **Mathematical operators:** Product, Sum, Gain, Min/Max, and any single-input/single-output function (Abs, Trigonometric, etc.) No logical blocks are allowed in the current implementation.
- **Signal and Systems:** Mux, Demux and ground.
- **Sources and Sinks:** Matlab workspace constant, In, Out, To workspace and From Workspace.
- **Nonlinear elements:** Controlled switch and Manual Switch.
- **Stateflow diagrams:** Hierarchical and concurrent.

The input models must comply with the following restrictions: (1) Stateflow diagrams can receive and provide continuous signals from and to Simulink. (2) Stateflow can also provide switching signals that are always connected to the control input of a Switch block. (3) Switches can be controlled only by these switching signals. These restrictions result in a clear separation of discrete and continuous behavior where all structural changes on the system are made through switches. Intuitively, each combination of these switches corresponds to a discrete location of the HA.

Example: Tank Level Control

To illustrate the steps a translation algorithm has to take, an example is provided in this section. As shown in Figure 49, there is a tank containing fluid, with an inlet pipe and two outlet pipes. Each pipe has a valve, named V1, V2 and V3 that can be in either open or closed state. A valve is modeled as a switch in MSS. Sensors can sense the height

of fluid in the tank (h) and the flow through valve V3 (em flow). A controller regulates the system using the state machine shown in the Figure 49. In the initial state of the system V1 is closed and V2 is open. When the height of the tank goes above 10 units then outlet values V1 and V3 are opened. When the flow through V3 becomes greater than 5 units, the inlet value V2 is closed. The inlet V2 is opened and outlet V1 is closed when the fluid level drops below 8 units.

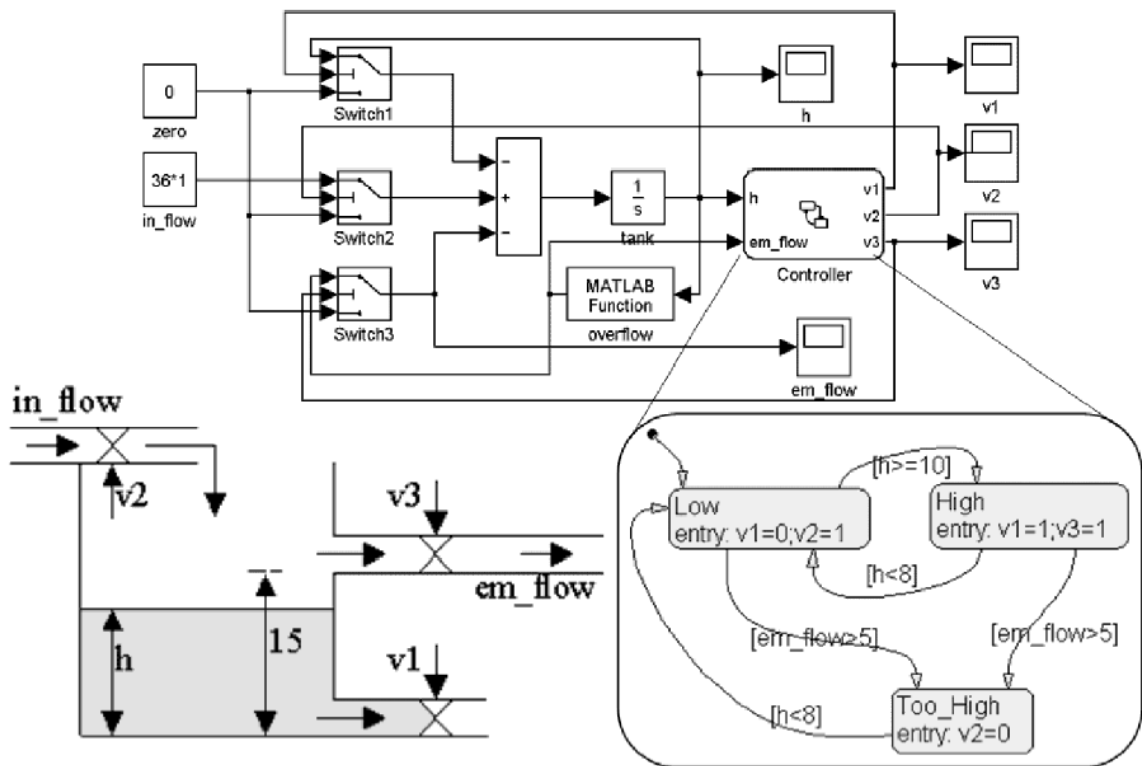


Figure 49 A tank with three valves

Looking at the models, the number of locations in the final hybrid automata is not apparent. On closer inspection it is seen that in the initial state Low, valve V1 is closed and V2 is open. However, the value of value V3 is unspecified, thus the initial state has discrete behavior, represented by the opening or closing of V3. Thus, state Low

needs to be split into two states such that one of the states is active when V3 is open, while the other one is active when the V3 is closed, connected via a state transition. Having inspected the entire system and the controller's state machine, the resulting state machine diagram can be drawn up as shown in Figure 50.

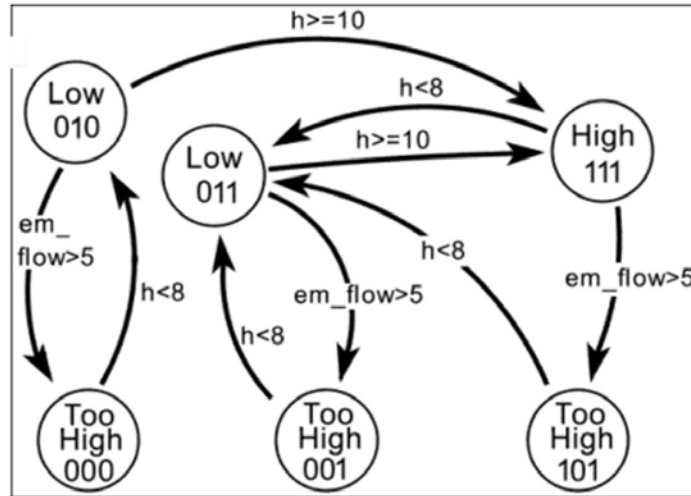


Figure 50 The "true" (hybrid automata) state machine for the tank example

After all the discrete states are identified, the next step is to find the differential equations for each state. Since the value of the switches are all defined for a given state, the Simulink diagram is now purely continuous and variable substitution can be used to find the differential equations. Differential equations are calculated from the output of the integrator block (see block with 1/S in Figure 50). For example, for location High111 in Figure 50, the differential equation for the tank (block 1/S in Figure 50) block can be found as follows. Let the output of each block have the same name as the block. Then, $d/dt(\text{tank}) = \text{Sum}$, where Sum is the output of the summation block that can be substituted with the sum of its inputs: $d/dt(\text{tank}) = (-\text{Switch1} + \text{Switch2} - \text{Switch3})$ Since the settings of the switches for this location are known, those paths will be chosen. A switch

having the value of '1' indicates that the topmost input of the switch is passed through. Thus, Switch1 will be replaced by the tank variable. Switch2 is replaced by 36*1 and Switch3 is replaced by the output of the MATLAB function which is 3*max(0,tank-15). Finally the differential equation of the tank level is:

$$\frac{d}{dt}(tank) = -tank + 36 - 3 * \max(0, tank - 15)$$

Implementing the Algorithm in GReAT

This translation algorithm has been implemented using GReAT. It contains 131 rules, 40 compound rules and 22 test/cases. The implementation is divided into two parts, the first deals with finding all the discrete locations in the Simulink/Stateflow diagram and the second deals with inferring the continuous dynamics for each location.

Translating Stateflow

In the Stateflow part of the algorithm (see Figure 51), first the Stateflow models are converted into an internal representation in CreateHierarchicalStateChart. Next, the hierarchical concurrent state machine is converted to its equivalent, "flat" finite state machine in HSM2FSM. Then in CreateVarAs, associations of Simulink switches with the states are transferred to the flat machine. At this stage StateSplitting, the splitting algorithm is performed (see Appendix F). After all the required discrete states/locations have been found, Reachability is executed that performs reachability analysis on the models to eliminate all unreachable states. At this state the number of discrete states in the system is known and corresponding locations in HSIF are created.

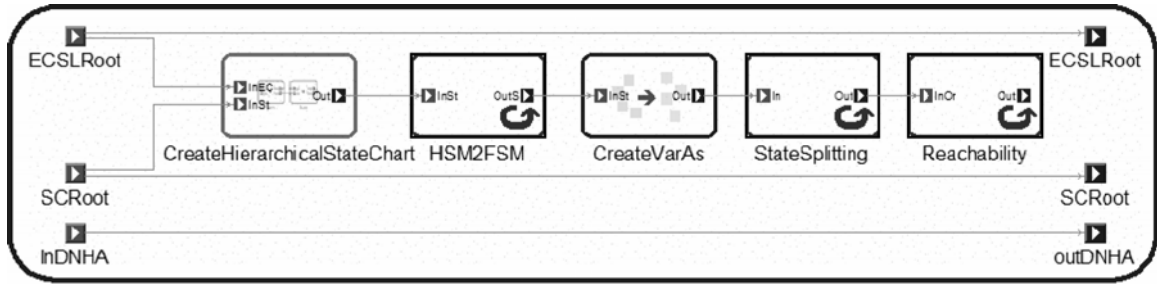


Figure 51 The StateflowPart Rule

HSM2FSM is the part of the transformation that converts a hierarchical concurrent state machine to an equivalent flat representation. The flattening algorithm is depth-first/bottom-up and is achieved using a recursive block *Top-level* (shown in Figure 52). *HSM2FSM* gets input from the input port *InState*. The input can be of type *or-state*, *and-state* or *simple-state*. The first expression inside *top-level* is a test/case called *Test* that branches according to the type of input. If the input is an *and-state* it is passed to the block called *And* that flattens the *and-state*. If the input is an *or-state*, it is passed to the block called *Or* that deals with the flattening the *or-state*, and if the input is a *simple-state* it is passed directly to the output port *OutState* without any processing.

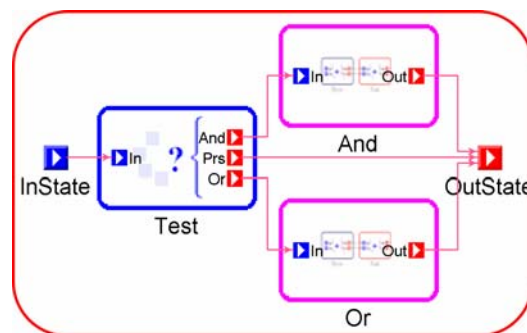


Figure 52 The HSM2FSM rule

Figure 53 shown the rules inside the *Or* block of Figure 52. These internal rules are used to flatten an *or-state*. The first rule in the rule chain is *CallRecursiveOnChildren*, a block that finds all the contained states of the *or-state* being processed and then called the *HSM2FSM* rule (Figure 52) for each of them. The next expression *TestForChild* will only execute after the recursive calls have been executed and thus at this point the *or-state* being flattened will only contain flat *or-states* (*and-state* when flattened will also produce an equivalent flat *or-state*) and primitive states. *TestForChild* is a test/case and it tests to see if the *or-state* contains any *or-state* type children. If not, then the *or-state* is already flat and is passed to the output port. If the *or-state* contains other *or-states* then it is passed to *ElevateChildOr* rule (Figure 54).

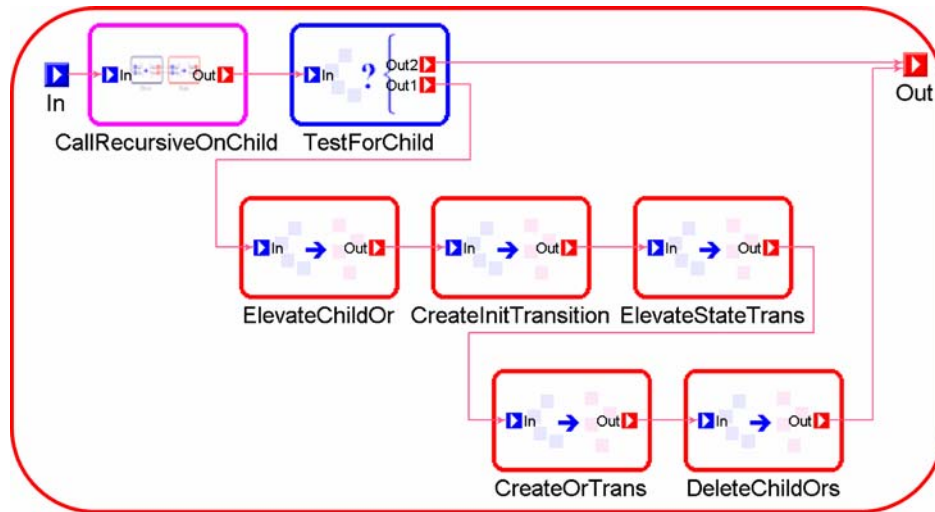


Figure 53 Inside the OR rule

Figure 54 shows *ElevateChildOr* rule. In the rule, the *or-state* being flattened is *Or1* and for each contained *Or1x* child *or-state* having a child *State*, a new *StateNew* is created as the child of *Or1*. The next rule in sequence is *CreateInitTransition*. This rule is used to create equivalent transitions for the init transition within *Or1*. *ElevateTrans* is the

next rule and it creates transitions for each transition contained in Or1x. *CreateOrTrans*
 The next rule is used to create equivalent transitions for each transition that is incident upon Or1x. The last rule in the sequence *DeleteChildOrs* is used to delete Or1x. At this stage the Or1 state is a flat or state.

Flattening an *and-state* is more complex and requires a few more rules. For the sake of brevity it has not been described here.

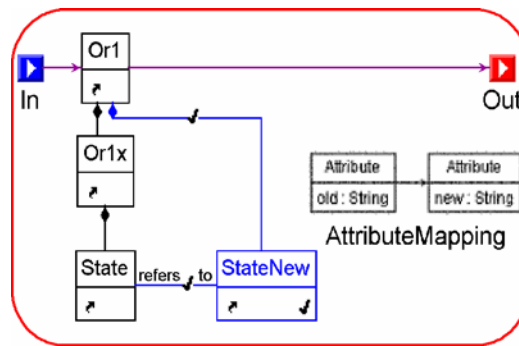


Figure 54 *ElevateChildOr* rule

StateSplitting (see Figure 55) is one of the most complex parts of the mapping and it is done in stages. The first stage is Infer Implicit Signals and it implements Step 2 of the algorithm described in Appendix F. This is followed by NewMachine which creates an empty state machine. The Create State Tribes performs state splitting based on Step 1. The next step is Transfer Transitions which implements Step 3 by appropriately mapped transitions to the new machine. If the initial state was split, an initial state is selected according to Step 5 in CreateInit. CarryBlockRef and In2Out perform housekeeping operations at the end.

The Infer Implicit Signals block in Figure 55 is performed repeatedly. In every iteration step, for every state the SetImplicitValue rule (see Figure 56) is called. In the

SetImplicitValue block all switching signals with color red are chosen. If there is an incoming transition which alters the state of the signal, then the transition is used to infer the new state of the signal. The translator will iterate until none of the signals change during a run, i.e. the iteration reaches a fixpoint.

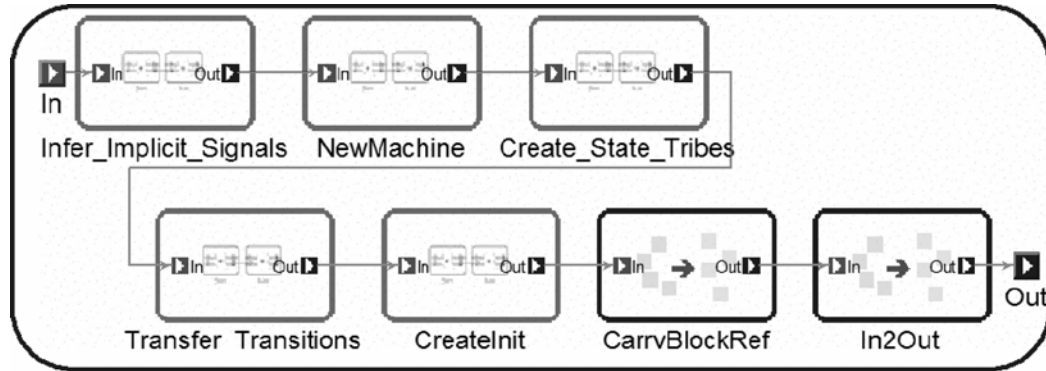


Figure 55 The StateSplitting rule

There are two main cases that can change the default interpretation of switching signal values. The first case is shown in Figure 56. For a given State and switch variable (called Data in the diagram), if there exists another state (OtherState) with a transition to State, OtherState may influence the value of Data. Each state has a relation with Data, and the relation has two attributes: color and value. Color can be either black or red, black implying that the state is set to the value, while red implying that the value was inferred. Value can be 0, 1, ?, X, where '?' specifies that the state does not influence data, while 'X' specifies that the state can set the data to either '0' and '1'.

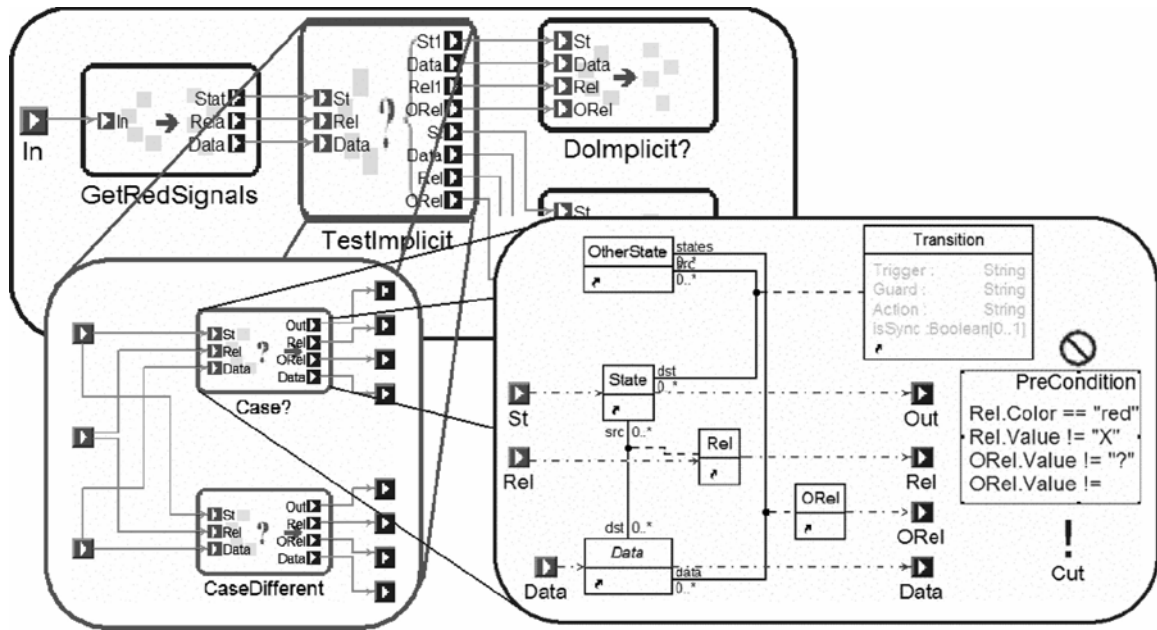


Figure 56 The SetImplicitValues Rule

In *Case?* if State's relation with Data is 'red' and value is not 'X' and OtherState's relation with the Data is '?' then, the inference is that the value of the current state's relation with data is also '?'. In *CaseDifferent* if OtherState's relation with Data is not '?' and is not the same as State's relation with Data. In this case the State's relation with Data is altered according to the following rules. If State's relation was '?' then it will take OtherState's relation. If State's relation is not the same as OtherState's then it will take the value of 'X'.

Implementation of the reachability analysis (see Figure 51) is based on the mark and sweep algorithm [102]. The algorithm starts with the init state and marks all the states it can find. Once all reachable states have been marked the second part traverses through the states and deletes all the states that have not been marked.

Translating Simulink

After all the states of the hybrid automata have been created, the next step is to identify the algebraic and differential equations for each location (Step 4 of Appendix F). The various steps in this translation are (1) identification of state variables, (2) identification of input and output variables (3) discovery of algebraic equations for dependent variables and (4) discovery of the differential equations for the state variables.

Each integrator block in Simulink is assigned a state variable. Each input port to the entire system becomes an input variable. Each source block of Simulink also becomes an input variable. Sink blocks and output ports become output variables. Some intermediate variables are created for interfacing with Stateflow. These variables depend on other independent variables in the system.

After all the variables have been identified, the next step is to determine algebraic equations of dependent variables and differential equations for state variables. These equations are location dependent, thus for each location the differential and algebraic equations are inferred using a backward trace algorithm. Starting from the Simulink port corresponding to the variable a backward trace is used to determine the blocks that provide input to the block. For each such block the block's type determines the kind of sub-expression the block will add to the equation (see Table 1). The back trace yields a tree with the termination points being state variables, input variables and constants.

Table 4 Mapping Simulink blocks to sub expressions

Simulink Block Type	#in	#out	Corresponding Sub expression
Sum	2..*	1	$(\pm S_1 \pm S_2 \pm S_3 \pm \dots \pm S_n)$
Mult	2..*	1	$(S_1 * S_2 * S_3 * \dots * S_n)$
Switch	2+1	1	Chose path based on switch position in given location
Gain	1	1	$(G * S)$
Min/Max	2..*	1	$\text{Min}(S_1, S_2, S_3, \dots, S_n) /$ $\text{Max}(S_1, S_2, S_3, \dots, S_n)$
Single Input Function (Abs, Signum/ Saturate)	1	1	$\langle \text{function name} \rangle (S)$
Integrator	1	1	$\langle \text{integrator variable name} \rangle$
Constant	0	1	$\langle \text{constant value} \rangle$

Translating the Tank Level Control example

This section shows how the algorithm described earlier can be used to translate the Simulink/Stateflow example described in Figure 49. Initially, in state Low, the value of V3 is undefined while the value of V2 is undefined in state High. In state Too High the value of V1 and V3 is undefined. After running the Infer Implicit Signals block there are some implicit values for undefined variables (see Figure 57(b)). For example, in state Low, the value of V3 can be both 0 and 1, while in state High the value of V2 was set to 1. After we determine the value of the switches in each state we can split the states that have switches with undefined values. In this example, the state Low will be split into two while the state Too High will be split into four new states (see Figure 57(c)).

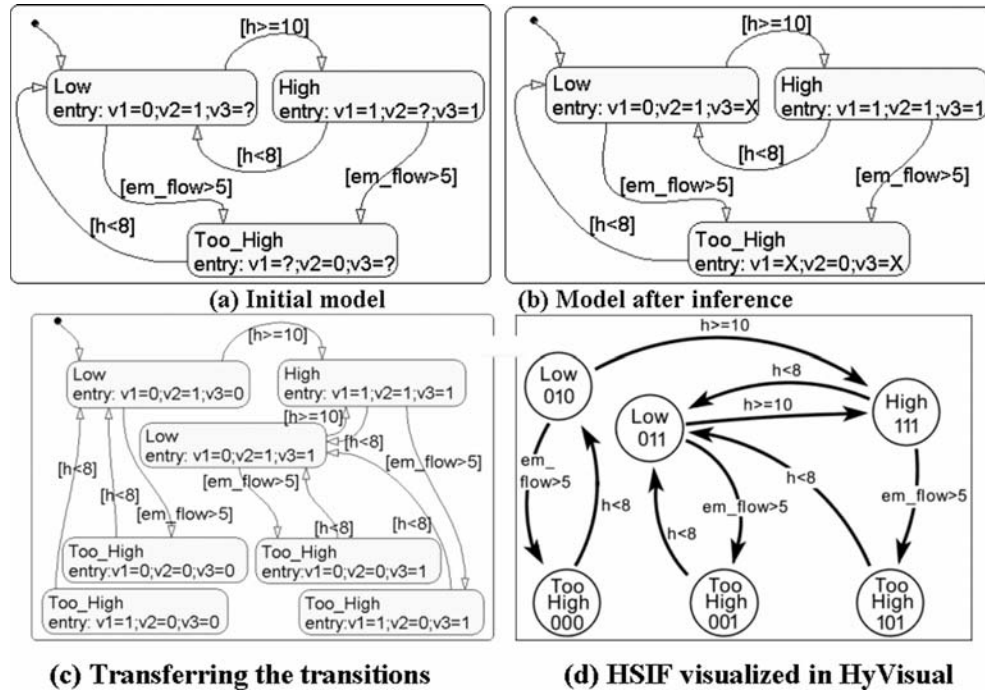


Figure 57 Stages of Stateflow splitting

After the states are split, transitions from the original machine need to be transferred to the new larger machine. The algorithm takes care of mapping the transitions correctly. After the equivalent machine is created, reachability analysis is performed. The analysis will reveal that state Too High with value of $V1 = 1$, $V2 = 0$ and $V3 = 0$ will never occur and it can thus be eliminated. Figure 57 (d) shows the locations in HSIF. The visualization is provided by HyVisual [100]. After all the discrete locations have been identified, the continuous time dynamics for each location will be found using the backward trace algorithm.

Summary

This Chapter described the implementation of an algorithm to convert MSS models into HSIF models. The MSS models may contain continuous time blocks,

Stateflow blocks, and switches, while the resulting HSIF model consists of a hybrid automaton that exhibits the same dynamic behavior as the original MSS model. This transformation demonstrates the capabilities of GReAT as a transformation language and shows how it can be used to solve complex real world problems.

One notable aspect of this implementation is that external support was not required for the implementation of the algorithm. Also all the different algorithms and graph manipulations required by the algorithms such as: state splitting, state space cross product, back trace of the Simulink graph, were achievable and easy to implement in GReAT.

During the development of the solution, the strengths and weaknesses of GReAT were identified. The strengths are:

1. Specification of structural manipulations was very easy and intuitive with GReAT.
2. There were fewer errors in the specification and debugging and finding the error was easier because of a visual representation.
3. Understanding the implementation after a long break (2 weeks to a month) was easier than it is in a regular programming language such as C++
4. Having a hierarchical representation helped significantly in managing the complexity of such large and complex transformation.

Some of the weaknesses of GReAT were:

1. There is no elegant way to parse string attributes and complex code needs to be written in the attribute mapping area.
2. Creation of a number of objects cannot be visually specified based on the information provided in attributes. Users need to write attribute mapping code to achieve this.
3. Since the transformation language is similar to functional programming, the entire required context need to be carried along and passed to all the intermediate rules.

Conclusion

The conclusion that can be drawn from this Chapter is: GReAT language is suitable for model-to-model transformations and it can be used to specify and automatically implement large complex models. It allows the user to write the transformation without worrying about implementation details such as accessing and manipulation models, executions of the pattern matcher and other similar issues. GRE and GRD provide an easy way to prototype and debug the transformations while the CG provides an efficient implementation once the transformation algorithm is fixed. The intuitive feeling is that efficiency of the developer is enhanced with the help of GReAT and the associated tools.

CHAPTER VII

RESULTS, CONCLUSIONS AND FUTURE WORK

Results

This section will evaluate GReAT and its tool suite with respect to the requirements defined in Chapter III. The evaluation should lead to the validation or the negation of the hypothesis. First, the requirements are revisited to see whether they were satisfied by GReAT.

Requirement 1

“The transformation language should have a sub-language for the specification of graph domains.”

GReAT uses UML class diagrams as the sub-language for the specification of graph domains.

Requirement 2

“The domain specification language should use a well know language or be based on one.”

UML class diagrams is a well known language which has been standardized [3]

Requirement 3

“The transformation should use the type information from the domains to strongly type the transformations.”

The pattern specification of GReAT uses the type information from class diagrams to make pattern specification and transformation strongly typed.

Requirement 4

“Often rewriting graphs belonging to one domain into graphs that belong to another domain is required.

- a. The language should support the specification of multiple domains.*
- b. It should have constructs that allow users to write rewritings where the input and output graphs are disjoint and do not even belong to the same domain.”*

In GReAT, any number of UML packages can be used. Each package contains a set of class diagrams that represent a domain. The user can specify new packages that are temporary and used only during the transformation. These packages can associate objects belonging to different packages. These temporary packages help integrate the independent packages only for the transformation. Thus, a graph rewriting problem can be treated as a transformation where the input graph is concentrated in one part of the domain while the output graph will be concentrated in a different part.

Requirement 5

“The computational power of the transformation language should be comparable to a Turing machine to ensure that any transformation conceivable can be handled by it.”

As mentioned in Chapter III, a Turing Machine (TM) is represented as a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where

$Q = \{q_0, q_1, \dots, q_m\}$ is a finite set of states

$\Sigma = \{s_1, s_2, \dots, s_n\}$ is a finite set of symbols called the input alphabet

Γ is a super set of Σ is a finite set of symbols called tape symbols

q_0 is an element of Q is the initial state

$\delta : Q \times (\Gamma \cup \{\Delta\}) \rightarrow (Q \cup \{h\}) \times (\Gamma \cup \{\Delta\}) \times \{R, L, S\}$ is a transition function

Here Δ denotes a blank and R, L and S denote move the head right, left and do not move it, respectively and h denotes the halt state. The tape symbol on the right side of the transition function is written to the current cell.

To prove that GReAT is Turing complete it needs to be established that any Turing machine can be converted to a GReAT program. The TM can be emulated in GReAT using a Domain for representing TMs.

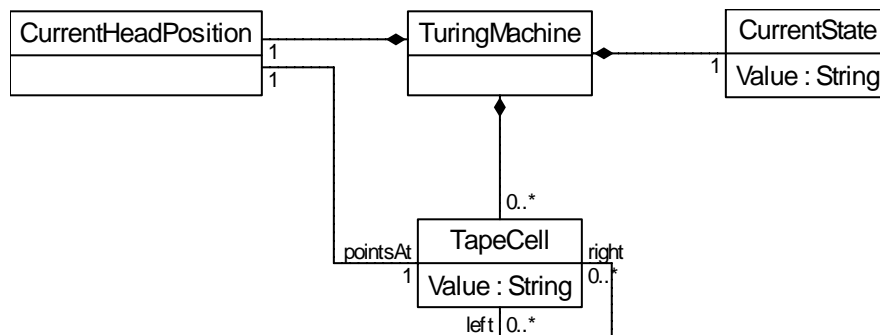


Figure 58 The domain of Turing machines

Figure 58 shows a domain that can store the necessary information required for implementing a TM. In the figure we see that a *TuringMachine* can contain an infinite number of *TapeCells*. These cells are organized such that each cell has a left cell and a right cell with the constraint that disallows a circular tape. The *TuringMachine* also contains one instance of the *CurrentState* object that stores the current state of the

machine. The *CurrentHeadPosition* object has an association to a *TapeCell* which describes the head position.

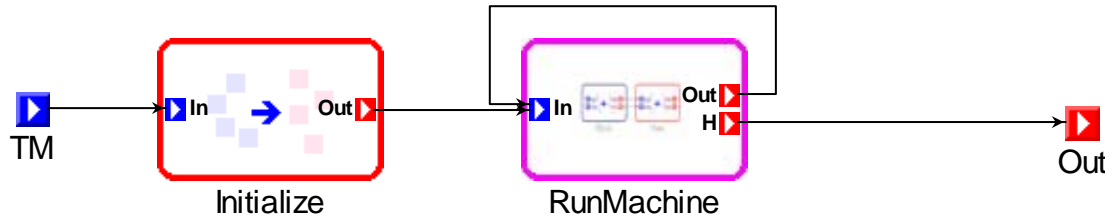


Figure 59 The top-level rule of Turing machine

The transition function can be represented as a GReAT transformation. At the top level (see Figure 59) the input Turing Machine is initialized with the *CurrentState.Value* field equal to the initial state. Then the machine is run till it reaches the halt state.

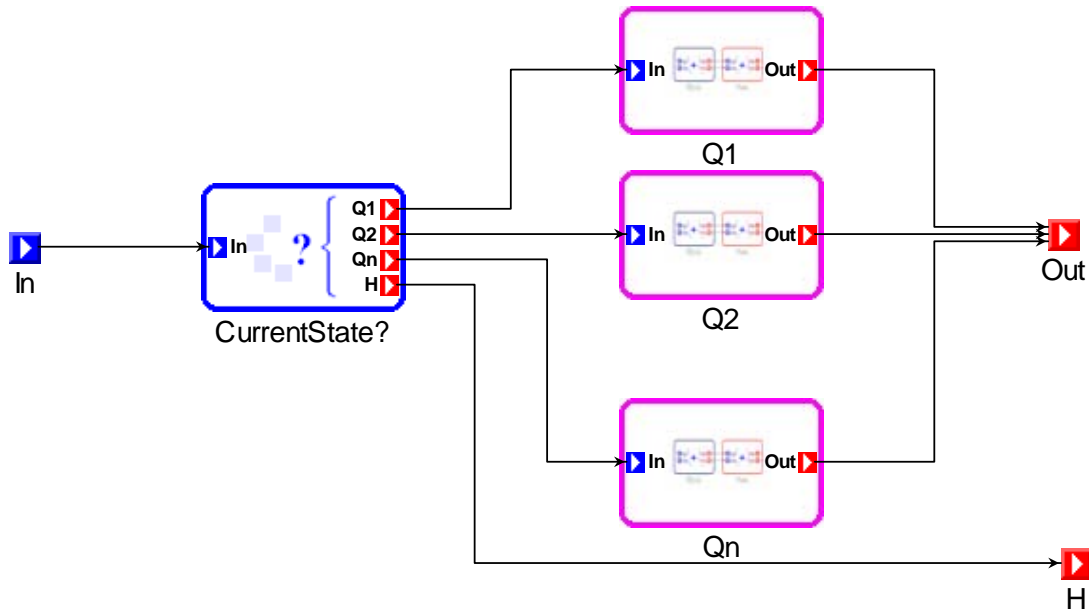


Figure 60 Internals of *RunMachine*

Figure 60 shows the *RunMachine* Block. First, a *Test* is used to determine the current state. Based on the current state, the block corresponding to it is used. In the block, action is taken and the current state is changed. Then the TM is passed out from the out port which is fed back to *RunMachine* (see Figure 59). If the current state is the halt state then the output goes to H output port which then terminates the program.

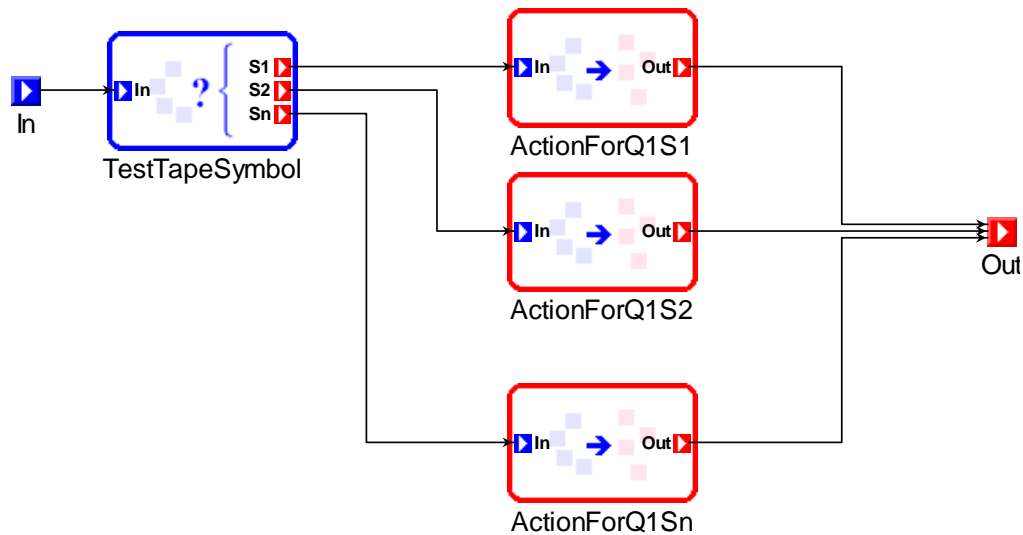


Figure 61 Inside Q1 block, choosing action for current state and symbol

Inside the block for a particular state there is a test for checking the current symbol. Based on the current symbol, a rule is taken where the action for the state will be taken. For example, in the block *Q1* (see Figure 61) the test chooses between different actions. Within an action (see Figure 62) the value of the current tape cell can be changed, a new state specified and the new position of the tape head can be changed to either the left or the right.

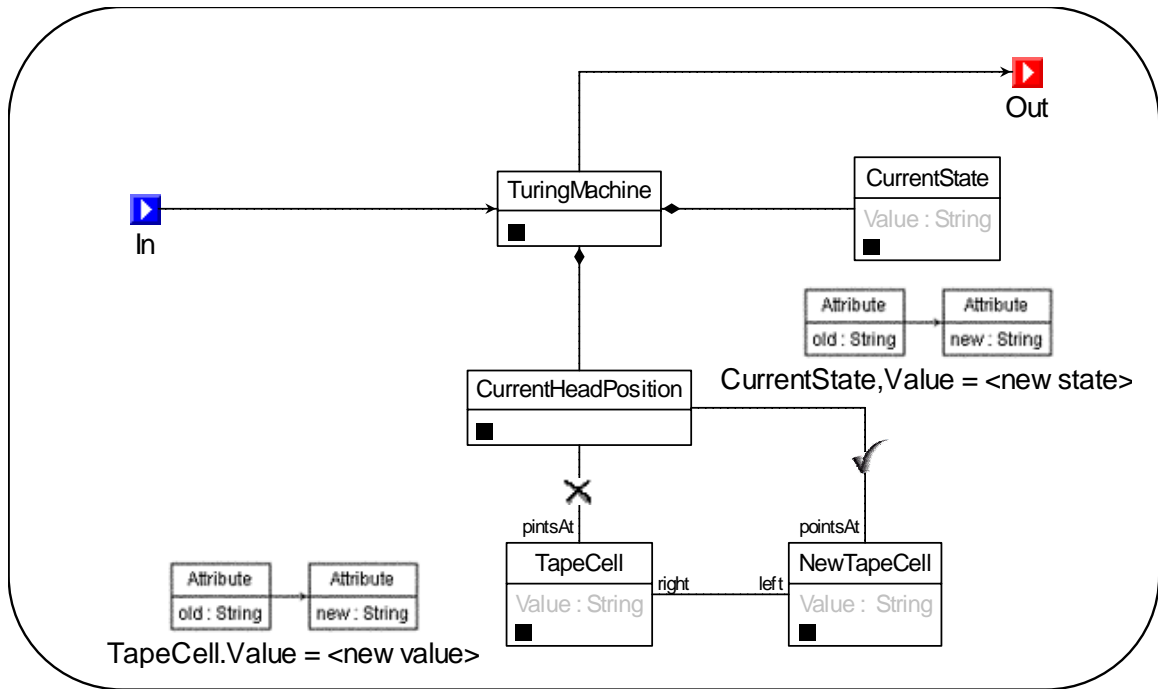


Figure 62 Action taken for a particular State, symbol pair.

Requirement 6

“The language should be capable of transforming/rewriting any number of graph/domain pair, not just two. There could be n input graphs and m output graphs and these graphs can belong to any number of domains.”

In GReAT any number of domains can be used and the transformation can work with an arbitrary number of input and output graphs. For example, the Simulink/Stateflow to HSIF translation works on three graphs belonging to different domains. The input graph is the Simulink/Stateflow graph. The Stateflow part is converted to an intermediate state machine representation called StateChart. Finally the output produced is a graph that belongs to the HSIF domain.

Requirement 7

“The language focus should be on constructs that allow users to write efficient transformations.”

Special attention was paid to the performance of the language constructs. As described in Chapter IV Section on optimized transformations, three language constructs have been described followed by details on how they can be used for building efficient transformations. The three techniques are (1) Typed Patterns, (2) Pivoted Patterns and (3) Reusing previously found objects.

Requirement 8

“The language should have efficient implementations of its programming constructs. The implementation should be comparable to its equivalent hand written code.”

Efficient algorithms and partial evaluation of different parts were used and the two primary methods for building efficient implementations of the language constructs. Appendix A, Appendix B and Appendix C are algorithms for efficient pattern matching using the notion of pivoted patterns. The CG, as described in Chapter V performs a partial evaluation of the generic pattern matching algorithms making the time constant much smaller.

Requirement 9

“The language should have a formal mathematical foundation that can facilitate the verification of transformations by theorem proving.”

Semantics of GReAT were defined in Object Z, a formal mathematical language used for expressing semantics (see Appendix D). This gives GReAT a strong

mathematical base. Given a transformation in GReAT, it can be converted into a set of operations and functions on the input and output domains. Proofs can then be written based on these transformations. Since the granularity of the transformations is much coarser than statements of a programming language, larger proofs should be possible.

As a demonstration of this capability a simple transformation problem is used. The transformation is required to produce an isomorphic copy of given graph. For the sake of simplicity, a single domain is used. The domain has one type of vertex SV and one type of edge SE. A temporary domain with one cross-link TempE from SV to SV is used. As shown in Figure 63 the transformation consists of three rules. The first rule *CopyVertices* creates a new vertex for every vertex in the graph and creates a TempE edge between them. The second rule *CopyEdges* creates the edges corresponding to the original graph, and the third rule, *DeleteOld* deletes the old graph.

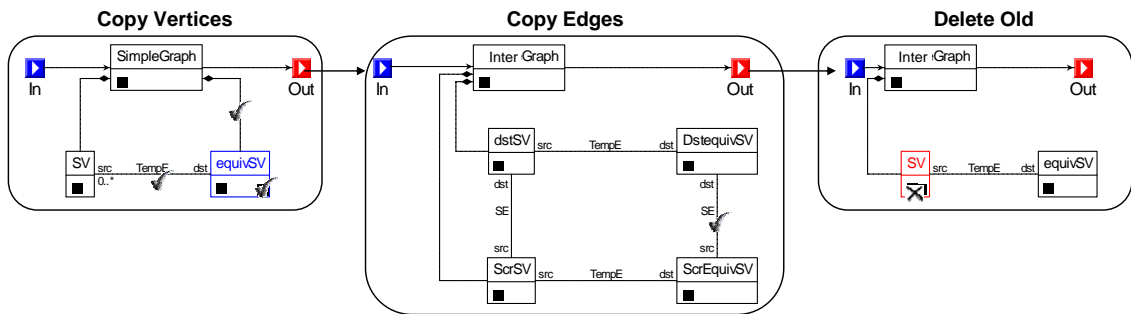


Figure 63 Transformation to make isomorphic copy of graph

In order to prove that the transformation actually performs an isomorphism, a formal proof is required. First we need to define graphs, their domains and the written the properties of the transformations.

Def 1: We call a set G a graph if

- (1) $G = \langle V, E, T_V, T_E, Src, Dst, VType, EType \rangle$ for some sets V, E, T_V, T_E called vertices, edges vertex types and edge types respectively and for some functions $Src, Dst, VType, EType$ called source vertex, destination vertex, vertex type and edge type.
- (2) $Src: E \rightarrow V \because Src(e) = v \in V$ for all $e \in E$
- (3) $Dst: E \rightarrow V \because Dst(e) = v \in V$ for all $e \in E$
- (4) $VType: V \rightarrow T_V \because VType(v) = t \in T_V$ for all $v \in V$
- (5) $EType: E \rightarrow T_E \because EType(e) = t \in T_E$ for all $e \in E$

Def 2: We call a graph I an intermediate graph if

- (1) $I = \langle V, E, T_V, T_E, Src, Dst, VType, EType \rangle$
- (2) $T_V = \{SV\}$
- (3) $T_E = \{SE, TempE\}$

Def 3: We define I as the class of all intermediate graphs

Def 4: We call an intermediate graph S a simple graph if

- (1) $S = \langle V, E, T_V, T_E, Src, Dst, VType, EType \rangle$
- (2) $EType(e) = SE$ for all $e \in E$

Def 5: We define Σ as the class of all simple graphs

Def 6: We define the operation $CopyVertices:\Sigma \rightarrow I$ where

- (1) $CopyVertices(S_{in}) = I_{out}$ for all simple graphs
- (2) $S_{in} = \langle V_{in}, E_{in}, T_V, T_E, Src_{in}, Dst_{in}, VType_{in}, EType_{in} \rangle$
- (3) $I_{out} = \langle V_{out}, E_{out}, T_V, T_E, Src_{out}, Dst_{out}, VType_{out}, EType_{out} \rangle$
- (4) $V_{out} \supseteq V_{in} \wedge E_{out} \supseteq E_{in}$
- (5) $Src_{out} \supseteq Src_{in} \wedge Dst_{out} \supseteq Dst_{in} \wedge VType_{out} \supseteq VType_{in} \wedge EType_{out} \supseteq EType_{in}$
- (6) $\exists FV \subseteq V_{in} \times (E_{out} - E_{in}) \times (V_{out} - V_{in})$
- (7) $(\forall v_{in} \in V_{in})(\forall e_t \in E_{out} - E_{in})(\forall v_{out} \in V_{out} - V_{in})$
 $(EType(e_t) = TempE \wedge Src(e_t) = v_{in} \wedge Dst(e_t) = v_{out}) \leftrightarrow ((v_{in}, e_t, v_{out}) \in FV)$
- (8) $(\forall v_{in} \in V_{in})(\exists!(v_{in}, e_t, v_{out}) \in FV)$
- (9) $(\forall v_{out} \in V_{out} - V_{in})(\exists!(v_{in}, e_t, v_{out}) \in FV)$
- (10) $(\forall e_t \in E_{out} - E_{in})(\exists!(v_{in}, e_t, v_{out}) \in FV)$

Def 7: We define the operation $CopyEdges:I \rightarrow I$

- (1) $CopyEdges(I_{in}) = I_{out}$ for all intermediate graphs and
- (2) $I_{in} = \langle V_{in}, E_{in}, T_V, T_E, Src_{in}, Dst_{in}, VType_{in}, EType_{in} \rangle$
- (3) $I_{out} = \langle V_{out}, E_{out}, T_V, T_E, Src_{out}, Dst_{out}, VType_{out}, EType_{out} \rangle$
- (4) $V_{out} = V_{in} \wedge E_{out} \supseteq E_{in}$
- (5) $Src_{out} \supseteq Src_{in} \wedge Dst_{out} \supseteq Dst_{in} \wedge VType_{out} \supseteq VType_{in} \wedge EType_{out} \supseteq EType_{in}$
- (6) $\exists FE \subseteq E_{in}^3 \times (E_{out} - E_{in})$
- (7) $(\forall e_{in}, e_{ts}, e_{td} \in E_{in}) \left(\begin{array}{l} EType(e_{in}) = SE \wedge EType(e_{ts}) = EType(e_{td}) = TempE \wedge \\ Src(e_{in}) = Src(e_{ts}) \wedge Dst(e_{in}) = Src(e_{td}) \end{array} \right) \rightarrow$
 $(\exists! e_{out} \in E_{out} - E_{in}) \left(\begin{array}{l} EType(e_{out}) = SE \wedge Src(e_{out}) = Dst(e_{ts}) \wedge \\ Dst(e_{out}) = Dst(e_{td}) \wedge (e_{in}, e_{ts}, e_{td}, e_{out}) \in FE \end{array} \right)$
- (8) $(\forall e_{in}, e_{ts}, e_{td} \in E_{in})(\forall e_{out} \in E_{out} - E_{in})$
 $((e_{in}, e_{ts}, e_{td}, e_{out}) \in FE) \rightarrow \left(\begin{array}{l} EType(e_{in}) = EType(e_{out}) = SE \wedge \\ EType(e_{ts}) = EType(e_{td}) = TempE \wedge \\ Src(e_{in}) = Src(e_{ts}) \wedge Dst(e_{in}) = Src(e_{td}) \wedge \\ Src(e_{out}) = Dst(e_{ts}) \wedge Dst(e_{out}) = Dst(e_{td}) \end{array} \right)$
- (9) $(\forall e_{out} \in E_{out} - E_{in})(\exists!(e_{in}, e_{ts}, e_{td}, e_{out}) \in FE)$

Def 8: We define the operation $DeleteOld: I \rightarrow \Sigma$ as

- (1) $DeleteOld(I_{in}) = S_{out}$ for all graphs and
- (2) $I_{in} = \langle V_{in}, E_{in}, T_V, T_E, Src_{in}, Dst_{in}, VType_{in}, EType_{in} \rangle$
- (3) $S_{out} = \langle V_{out}, E_{out}, T_V, T_E, Src_{out}, Dst_{out}, VType_{out}, EType_{out} \rangle$
- (4) $(\forall v_{in} \in V_{in})$
 $((\exists e_t \in E_{in})(Src(e_t) = v_{in} \wedge EType(e_t) = TempE)) \leftrightarrow (v_{in} \notin V_{out})$
- (5) $(\forall e_{in} \in E_{in})$
 $(EType(e_{in}) = SE \wedge Src(e_{in}), Dst(e_{in}) \in V_{out}) \leftrightarrow (e_{in} \in E_{out})$

Def 9: We define the operation $IsomorphicCopy: \Sigma \rightarrow \Sigma$ as

$$DeleteOld \circ CopyEdges \circ CopyVertices$$

Theorem 1: For all $G_{in} \in \Sigma$, G and $IsomorphicCopy(G)$ are isomorphic

Proof:

$$\begin{aligned} \text{Let } G_{in} &= \langle V_{in}, E_{in}, T_V, T_E, Src_{in}, Dst_{in}, VType_{in}, EType_{in} \rangle \text{ and} \\ CopyVertices(G_{in}) &= G_{i1} = \langle V_{i1}, E_{i1}, T_V, T_E, Src_{i1}, Dst_{i1}, VType_{i1}, EType_{i1} \rangle \\ CopyEdges(G_{i1}) &= G_{i2} = \langle V_{i2}, E_{i2}, T_V, T_E, Src_{i2}, Dst_{i2}, VType_{i2}, EType_{i2} \rangle \\ DeleteOld(G_{i2}) &= IsomorphicCopy(G) = G_{out} \\ &= \langle V_{out}, E_{out}, T_V, T_E, Src_{out}, Dst_{out}, VType_{out}, EType_{out} \rangle \end{aligned}$$

Lemma 1: $V_{out} = V_{i1} - V_{in}$

Proof:

1. For all v element of V_{out} , v is an element of V_{i2} since V_{i2} is a superset of V_{out} and v is an element of V_{i1} since V_{i1} is equal to V_{i2} and
2. For all v element of V_{out} v is not an element of V_{in} since for all v_{in} elements of V_{in} there exists a temporary edge such that v_{in} is the source of the edge (Def 6: (8)) and by Def 8: (4) vertices that are the source of a temporary edge cannot be an element of V_{out} .

3. For all v element of $V_{i1} - V_{in}$, v is not the source of a temporary edge (Def 6: (7)) and thus is an element of V_{out} (Def 8: (4)).

Def 10: We define a relation $IFV : V_{in} \rightarrow V_{out}$ as

$$((v_{in}, v_{out}) \in IFV) \leftrightarrow (\exists e_t)((v_{in}, e_t, v_{out}) \in FV)$$

We claim that IFV is a bijection. First recall that by Lemma 1, $V_{out} = V_{i1} - V_{in}$. By Def 6: (8), for each $v_{in} \in V_{in}$ there is a unique $v_{out} \in V_{i1} - V_{in} = V_{out}$ with $((v_{in}, v_{out}) \in IFV)$. By Def 6: (9), for each $v_{out} \in V_{i1} - V_{in} = V_{out}$ there exists a unique $v_{in} \in V_{in}$ with $((v_{in}, v_{out}) \in IFV)$. Thus, IFV is a 1-1 correspondence, i.e., a bijection.

Lemma 2: $E_{in} = E_{i1} - \{e \in E_{i1} \mid EType(e) = TempE\}$

Proof:

1. For all e element of E_{in} , e is an element of E_{i1} since E_{i1} is a superset of E_{in} .
2. For all e element of E_{in} e is not an element of $\{e \in E_{i1} \mid EType(e) = TempE\}$ since E_{in} is a simple graph and simple graphs cannot have temporary edges.
3. For all e element of $E_{i1} - \{e \in E_{i1} \mid EType(e) = TempE\}$, e is a SE and thus in E_{in} (Def 6: (7) and (10)).

Lemma 3: $E_{out} = E_{i2} - E_{i1}$

Proof:

1. For all e element of E_{out} , e is an element of E_{i2} since E_{i2} is a superset of E_{out} .

2. For all e element of E_{out} , e is not an element of E_{il} since for all e_{i1} of E_{i1} either the Src or Dst or both is the Src of temporary edges (Def 6: (7) and (8)) and vertices that are the Src of a temporary edge are not part of V_{out} .
3. For all e element of $E_{i2} - E_{i1}$, e is a SE (Def 7: (7)). Src and Dst of e are Dst of temporary vertices (Def 7: (7)). Thus, e is neither the Src nor the Dst of a temporary edge (Def 6: (7)) and thus is an element of E_{out} (Def 8: (5)).

Def 11: We define a function $IFE : E_{in} \rightarrow E_{out}$ as

$$((e_{in}, e_{out}) \in IFE) \leftrightarrow (\exists e_{ts}, e_{td}) ((e_{in}, e_{ts}, e_{td}, e_{out}) \in FE)$$

We claim that IFE is a bijection. First recall that by Lemma 3, $E_{out} = E_{i2} - E_{i1}$, and by Lemma 2, $E_{in} = E_{i1} - \{e \mid EType(e) = TempE\}$. By Def 7: (7), for each $e_{in} \in E_{i1} - \{e \mid EType(e) = TempE\} = E_{in}$ there is a unique $e_{out} \in E_{i2} - E_{i1} = E_{out}$ with $((e_{in}, e_{out}) \in IFE)$. By Def 7: (7), (8) and (9), for each $e_{out} \in E_{i2} - E_{i1} = E_{out}$ there is a unique $e_{in} \in E_{i1} - \{e \mid EType(e) = TempE\} = E_{in}$ with $((e_{in}, e_{out}) \in IFE)$. Thus IFE is a 1-1 correspondence, i.e., a bijection.

We claim that IFV and IFE are edge preserving. By Def 6: (7), for all vertices v in V_{in} there exists a temporary edge with v as the source. By Def 7: (8), for all simple edges $e_{in} \in E_{i1} - \{e \mid EType(e) = TempE\} = E_{in}$ there exists a unique simple edge with the source and destination vertices being the destination of temporary edges of the corresponding source and destination vertices. $IFE(e_{in}) = e_{out}$ implies that

$IFV(\text{Src}(e_{in})) = \text{Src}(e_{out})$ and $IFV(\text{Dst}(e_{in})) = \text{Dst}(e_{out})$. Thus *IFE* and *IFV* are edge preserving.

Thus, for all $G_{in} \in \Sigma$, G and $IsomorphicCopy(G)$ are isomorphic.

This example transformation and the accompanying proof show that GReAT can be used for specifying transformations and have a formal verification of the properties of interest. For larger and more complex transformations the definitions can get more complicated. The positive side of the transformation language is that it is conceivable to write a translator that can convert the transformation specification into mathematical definitions. The theorem proving can then be done either by hand or using different heuristics.

Revisiting the Research Hypothesis and Completion Criteria

“A Metamodel based transformation language using graph rewriting and transformations that support multiple graphs (that may belong to different domains) with an efficient implementation is suitable for the specification of model transformers. Such a language should help shorten the time taken to develop model transformers and allow for formal proof of correctness of the transformations.”

GReAT is a graph transformation based model-to-model transformation language that supports multiple input and output graphs. In the Simulink/Stateflow transformation the need for transformation that uses more than two domains was seen. Thus, the requirement for multiple domains has been justified. The research hypothesis claimed that a graph transformation based language with efficient implementation would help speedup the time taken to develop model transformations and would allow formal

theorem proving. The theorem proving aspect has been demonstrated with the help of an example.

The two completion criteria were (1) expressiveness and (2) usefulness of the language. Expressiveness is measured in two dimensions. The first metric is the class of problem the language can solve. For instance, if the language is Turing complete then it can compute solutions to all problems that a Turing machine can solve. The second metric is whether it can be used to solve real-world problems. It was found that GReAT satisfies both these requirements. It has been proven to be Turing complete and thus as powerful as any other programming language. More importantly, both challenge problems and a host of other real-world problems were solved using GReAT, demonstrating that it is actually practically usable.

The second completion criterion was the usefulness of the language. This is the measure of the effectiveness of expressing the class of problems it targeted. This should also give us some insight into the question of whether such a language would provide a speedup over other conventional approaches. From user experience (discussed in the summary section of Chapter VI) it was seen that GReAT offers some advantages over other approaches such as (1) It was easy of specifying structural manipulation, (2) Errors were caught early and were fewer, (3) Manageability of the complexity simpler with hierarchical decomposition and (4) Maintaining and enhancing the transformations was more convenient because understanding previous work was more intuitive. These observations help us believe that the language would provide a speedup. However, a lot of empirical data over many projects and people is required to establish such a claim. Some preliminary work has been done in this field to gather such data.

Table 5 shows a compilation of the transformation specified in GReAT, the complexity of the transformation in GReAT, the time taken and the lines of code it was or would have been in a traditional language. The table provides some insights into how the transformations relate to hand code. If we use some code time metrics to speculate the time it would have taken to write the code, we can get an approximate idea of the speedup achieved.

Table 5 Compilation of different projects developed in GReAT

Problems	GReAT		Hand code
	Primitive/ Compound Rules	Time (man- hours)	Est. LOC
Mark and sweep algorithm on Finite State Machine (FSM)	7/2	~2	~100
Hierarchical Data Flow (HDF) to Flat Data Flow (FDF)	11/3	~3	~200
Hierarchical Concurrent State Machine (HCSM) to Finite State Machine (FSM)	21/5	~8	~500
Simulink Stateflow to C code	70/50	~25	~2500
Matlab Simulink/ Stateflow to Hybrid System	154/43	~50	~5000

Another technique for establishing the merits of GReAT is to show the first class entities of the language and show how and why they may help provide a speedup. The list of new entities that have been made first class objects in GReAT are

1. A powerful pattern specification language which has elevated the specification of patterns to be matched as a first class entity. The built-

in pattern matching algorithms allow the users to simply specify the pattern and not have to worry about how to match it.

2. Graph transformation language that allows users to specify manipulation of the graphs in an intuitive manner.
3. Heterogeneous metamodel that allows users to specify all the temporary data structures in the same formalism as the source and target. It elevates temporary information to first class entities.
4. Controlled transformations provide the user with the facility to sequence the transformations, use test cases and other programming constructs. The language fuses both declarative and imperative constructs in a manner that is intuitive for users and helps them be more efficient.

The above mentioned list of features describes the various language components that help make the language suitable and useful for specifying model-to-model transformations.

Conclusion

Computing languages continue to evolve toward higher levels of abstraction. The journey has taken programming languages from machine code to state of the art languages for component-oriented systems. A survey of modelling languages and looking into future trends in software engineering revealed a trend towards domain-specific modelling languages that may have both textual and graphical notation. Earlier attempts at CASE tools and domain-specific languages were studied to identify the reasons for limited success. The conclusion was that developing custom domain-specific languages

suffered from the problems of (1) high cost, (2) lack of standardization and (3) robustness.

It was argued that these problems can be avoided by the using a framework approach to developing domain-specific languages. A set of requirements for such a framework were identified. The areas of Model Integrated Computing (MIC) and Generative Programming (GP) were studied where an attempt to make such a framework has already been made. The conclusion of the search was that MIC-based frameworks were more suitable for the specification of domain-specific languages. Thus different MIC based tools were evaluated to see whether they fit the requirements. All tools that were surveyed lacked a formal language for specifying the dynamic semantics of the domain-specific languages. This step was usually achieved by writing a model interpreter or compiler that implemented the semantics of translating the models to a known semantic framework.

This deficiency of MIC frameworks was identified as the key limitation and various approaches to solve the problem were studied. Since models can be represented as graphs, the field of graph grammar and transformation was studied. Graph transformations seem to be ideal for a model transformation language. Nevertheless, these approaches could not be used directly for model-to-model transformations, and this posed some interesting challenges. These challenges were as follows: (1) multiple graph domains may be involved in the transformation, (2) there is a need for specification and use of links that cross domains, and (3) support for sequencing the transformation rules are required. Due to these requirements previous approaches could not be directly used.

Based on the literature survey, a research hypothesis was made that argued that a graph transformation based language would be suitable for model-to-model transformations. The requirements for such a language were laid out and the completion criteria were defined.

Graph Rewriting and Transformations (GReAT): a graphical language that addressed these requirements was introduced. GReAT is based on the use of UML class diagrams (and OCL) for representing the domains of the transformations (and structural integrity constraints over those domains). Transformations over multiple domains were supported, and cross-links among domains were defined at the metamodeling level.

The transformation language itself was divided into three sub languages: (1) Pattern Specification language, (2) Graph Rewriting/Transformation language and (3) the language for Controlled Graph Rewriting and Transformation. The Pattern Specification language introduced a concise way to represent fairly complex graphs, and various pattern matching algorithms were also developed. The Graph Rewriting/Transformation language was used to define graph transformation steps. Pattern graphs were embellished with actions like *new*, *bind*, and *delete* to express actions within a transformation. Pre-conditions for the transformations were captured in the form of a guard, and attribute mappings were used to modify the values of attributes. The language for Controlled Graph Rewriting and Transformation defined high-level, hierarchical control structures for rule sequencing, modularization, and branching.

In order to realize GReAT as a usable language, a concrete syntax was given to it. The concrete syntax defined the concrete entities and their visualizations. An abstract syntax for GReAT was also designed with an XML representation, thus isolating the

tools from the concrete syntax. An execution engine called Graph Rewriting Engine (GRE) was developed for GReAT. The GRE could read a GReAT specification and execute it on a given input to produce output. A debugger called Graph Rewriting Debugger (GRD) was also developed on top of the engine to allow users to single step through the GReAT transformations. The debugger also provided visualization front-end to drive the debugger and visualize the objects at different times. The GRE and GRD are good for prototyping the verifying the correctness of the transformation but the execution speeds is not acceptable for deployment of the transformations. For this reason a Code Generator (CG) was also developed for GReAT. CG converts a GReAT specification into efficient C++ code that can be compiled to make a stand alone transformer.

Apart from all the execution engines, an IDE was also developed around GReAT. The IDE consisted of an editor for creating GReAT transformations and a suite of tools to help the user in this process. The tools accompanying the IDE are divided into three different categories (1) Development tools that help users build models, (2) Transformation tools that convert the front-end concrete syntax to the abstract representations and (3) Invocation tools that invoke GRE, GRD and CG from the environment and provide feedback.

A case study was also presented where GReAT was used to solve the translation problem from Simulink/Stateflow to HSIF. This translation was quite complex and required different algorithms that performed various graph traversals and manipulations. GReAT was able to specify all the components of the transformations and thus demonstrate that GReAT can be used to solve large real-world problems.

The results section evaluated whether GReAT successfully satisfies the requirements that were laid out in the proposal and if it was able to uphold the hypothesis. The conclusions were quite convincing as GReAT was able to satisfy all the requirements. Evaluation of the expressiveness of GReAT was demonstrated with the help of a Turing completeness proof and the example problems solved in it. The usefulness issue was a bit difficult as a lot of empirical data over large periods of time are required. The usefulness was demonstrated with the help of user experience, some empirical data and a listing of all the first class entities in GReAT that would help the translator developer. A simple transformation was used as an example to demonstrate that GReAT translations can be used to write formal proofs of correctness.

Future Work

GReAT is not the end but the beginning of a research direction. Future research in this area is divided into two main categories. First is the further development of GReAT into a mature graph transformation language that can be used not only in MDA or model-to-model transformation but also for the manipulation of typed multi-graphs. Data in various storage formats such as XML, MOF and databases can be considered as graphs, thus widening the scope and impact of GReAT. The current implementation of GReAT is similar to stateless functional languages. One short-term goal is to increase the usability of the language by investigating object oriented and component oriented constructs and evaluating how they may be used in the transformation language.

The second research direction is that of using GReAT, a formal language as the starting point for “correct by construction” languages where correctness properties are guaranteed on every sentence of the language. For example, a transformation language

that guarantees not to violate the structural properties of a graph can be called “structure preserving”. The next step will be a transformation language that will be “static semantics preserving”, and eventually it may be possible to develop transformations that are “property preserving”. That is, transformations written in the language are guaranteed to preserve graph properties or other domain specific properties.

The long term goal of the research is to develop a framework that enables the rapid development and deployment of robust domain-specific languages. Such a framework will need to support the specification and automated implementation of the abstract syntax, visualization, static semantics and dynamic semantics of a new language with robustness guarantees in a short delivery time.

APPENDIX A.

ALGORITHM FOR SINGLE CARDINALITY PATTERN MATCHING

```
Function Name      : PatternMatcher
Inputs            : 1. Pattern Graph pattern
                   : 2. Match p_match (a partial Match)
Outputs          : 1. List of Matches matches

matches = PatternMatcher (pattern, p_match)
{
    foreach pattern edge with valid binding for both Src and Dst vert
    {
        if(corresponding graph edge doesn't exists)
        {
            return an empty match list
            Bind pattern and host graph edge.
            Add binding to p_match
            Delete the pattern edge from the pattern
        }
    }
    Edge edge = get pattern edge with exactly one vertex bound
    if(edge exists)
    {
        vertices = host graph vertices adjacent to bound vertex
        make a copy of pater in new_pattern
        Delete edge from new_pattern
        foreach vertex v in vertices)
        {
            new_match = p_match + new binding(unbound pattern
                                              vertex, vertex)
            ret_match = PatternMatcher(new_pattern, graph,
                                      new_match)
            Add ret_match to matches
        }
        Return matches
    }
    If(all patern edges are bound)
    {
        Add p_match to matches
        Return matches
    }
    else
        Return empty list
}
```

APPENDIX B.

ALGORITHM FOR FIXED CARDINALITY PATTERN MATCHING

```
Function Name      : PatternMatcher
Inputs            :      1. Pattern Graph pattern
                   :      2. Match p_match (a partial Match)
Outputs          :      1. List of Packets matches

matches = PatternMatcher (pattern, p_match)
{
  new_pattern = copy of Pattern.
  foreach pattern edge with both Src and Dst vertices bound
  {
    if(corresponding edge doesn't exists between host graph vertices)
      return false.
    Add edge binding to p_match
    Delete edge from new_pattern.
  }

  Edge edge = pattern edge with one vertex bound to host graph
  if(edge exists)
  {
    Delete edge from new_pattern.
    foreach vertex v in bound vertices of edge
    {
      peer_vertices[v] = vertices adjacent to vetrex bound to v
    }
    Intersect all the peer_vertices to form new list peer
    If(cardinality of peer Ci >= Cd cardinality of corresponding
pattern vertex)
    {
      For(Each combination of Cd from Ci)
      {
        peer_c is the unique combination
        new_match = p_match + new binding(pattern vertex,
peer_c)
        ret_match = PatternMatcher(new_pattern, new_match)
        Add ret_matches to Matches
      }
      Return matches.
    }
  }

  If(all patern matches are bound)
  {
    Add p_match to matches.
    return matches.
  }
  else
    return empty list.
}
}
```

APPENDIX C.

ALGORITHM FOR VARIABLE CARDINALITY PATTERN MATCHING

Before defining the algorithm for Variable cardinality pattern matching the definitions in Chapter VI need to be extended with some new definitions.

Vertices Adjacency Table

Vertices Adjacency Table: A Vertices Adjacency Table vat is an ordered pair of a pattern vertex and a set of vertices adjacency.

$vat = (pv, VA)$, where $VA = \{va \mid va \text{ is a vertices adjacency}\}$,

VAT Functions

$GetAdjVertices: VAT \times PV \times PE \times VV \rightarrow AVVV$

$\forall vat \in VAT, \forall pv \in PV, \forall pe \in PE, \forall V \in VV, GetAdjVertices(vat, pv, pe, V)$
 $return AVV \mid va \in VA, va = (pe, V, AVV) \wedge vat = (pv, VA)$

Some Additional Functions

$CreateVerticesAdjacencyTablesForPatternVertex : PV \rightarrow VAT$

$\forall pv \in PV, CreateTableForEachPatternVertex(pv) =$
 $n = \{v \mid v \in V (\text{host graph vertices that can bind with } pv)\}$

$\forall V = {}^n C_r, 1 \geq r \geq c, c = Cardinality(pv)$

$\forall pv_{adj}$ adjacent to pv

$V_{adj} = \{v \mid v \text{ is a set of host graph vertices that are adjacent to } V \text{ and can bind with } pv_{adj}\}$

$va = \{V_{adj} \mid \forall V_{adj} \text{ described above}\}$

$VA = \{va \mid \forall va \text{ described above}\}$

$vat = (pv, VA)$

return vat

Variable Cardinality Algorithm

MatchDynamic : $PG \times HG \times M \rightarrow \text{set of } M$

```
 $\forall pg \in PG, \forall hg \in HG, \forall m \in M, \text{MatchDynamic}(pg, hg, m) =$   
e = EdgeWithBothVerticesBound(m)  
if e  $\neq \varnothing$   
then  
    MatchDynamicBothBound(pg, hg, m, e)  
else if (e = EdgesWithSingleVertexBound(m))  $\neq \varnothing$   
then  
    if Src(e) is bound  
    then  
        MatchDynamicSrcBound(pg, hg, m, e)  
    else  
        MatchDynamicDstBound(pg, hg, m, e)  
    end if  
else if (e = EdgesWithNoneVerticesBound(m))  $\neq \varnothing$   
then  
    MatchDynamicNoneBound(pg, hg, m, e)  
else if all pattern vertices and edges are bound  
then  
    add match m to return set of matches  
end if
```

MatchDynamicBothBound: $PG \times HG \times M \times PE \rightarrow \text{set of } M$

$\forall pg \in PG, \forall hg \in HG, \forall m \in M, \forall pe \in PE, MatchDynamicBothBound(pg, hg, m, pe) =$
 $V_{Src} = GetHost4Pattern(m, Src(pe))$
 $V_{Dst} = GetHost4Pattern(m, Dst(pe))$
 For each $v_{Src} \in V_{Src}$
 For each $v_{Dst} \in V_{Dst}$
 If $\neg \exists E \subset Edges(hg) \mid \forall e \in E, Src(e) = v_{Src} \wedge Dst(e) = v_{Dst}$
 Then return false
 eb = (pe, unoin of all E)
 add eb to EB where m = (VB < EB)
 call MatchDynamic(pg, hg, m)
 delete eb from EB where m = (VB, EB)
 return

MatchDynamicSrcBound: $PG \times HG \times M \times PE \rightarrow \text{set of } M$

$\forall pg \in PG, \forall hg \in HG, \forall m \in M, \forall pe \in PE, MatchDynamicSrcBound(pg, hg, m, pe) =$
 $V_{Src} = GetHost4Pattern(m, Src(pe))$
 for each $V_i \subseteq V_{Src}$
 $V_{iDst} = GetAdjVertices(vat, Src(pe), pe, V_i)$
 For each $V_{jDst} = GetAdjVertices(vat, Src(pe), pe, V_j)$, where $V_j \subset V_i \subseteq V_{Src}$, if $V_{iDst} \not\subset V_{jDst}$
 $E_{Src2Dst} = \{e \mid e = (name, Type(pe), v_{Src}, v_{dst}), v_{Src} \in V_{Src} \wedge v_{dst} \in V_{jDst}\}$
 eb = (pe, $E_{Src2Dst}$)
 vb = (Dst(pe), V_{iDst})
 add eb to EB where m = (VB, EB)
 add vb to VB where m = (VB, EB)
 MatchDynamic(pg, hg, m)
 delete eb from EB where m = (VB, EB)
 delete vd from VB where m = (VB, EB)

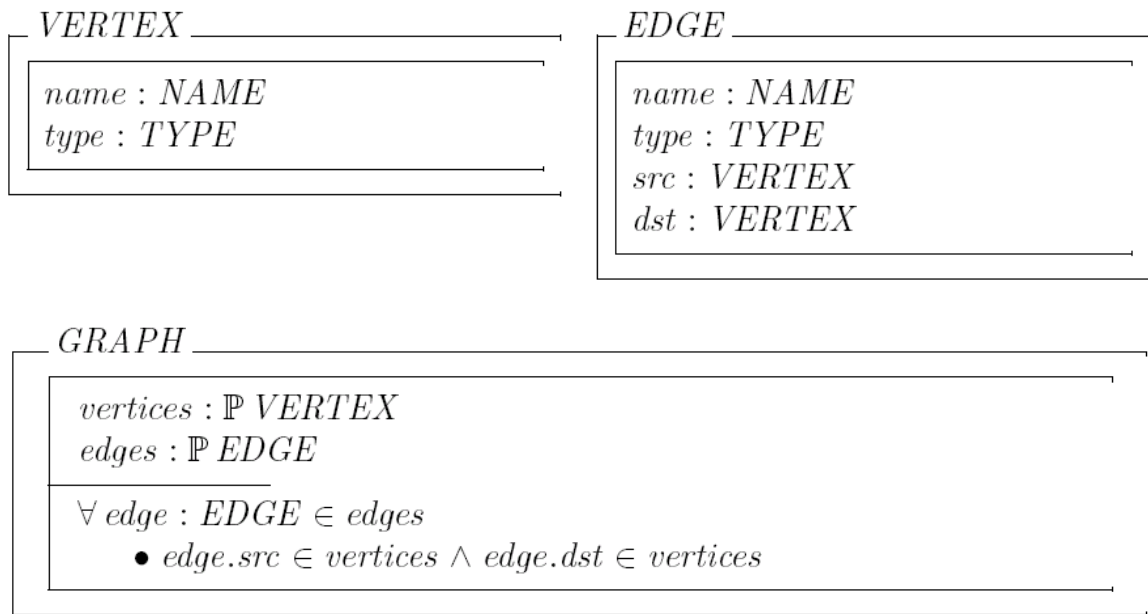
MatchDynamicDstBound: $PG \times HG \times M \times PE \rightarrow \text{set of } M$

$\forall pg \in PG, \forall hg \in HG, \forall m \in M, \forall pe \in PE, \text{MatchDynamicDstBound}(pg, hg, m, pe) =$
 $V_{Dst} = \text{GetHost4Pattern}(m, \text{Dst}(pe))$
 for each $V_i \subseteq V_{Dst}$
 $V_{iSrc} = \text{GetAdjVertices}(\text{vat}, \text{Dst}(pe), pe, V_i)$
 For each $V_{jSrc} = \text{GetAdjVertices}(\text{vat}, \text{Dst}(pe), pe, V_j)$, where $V_j \subset V_i \subseteq V_{Src}$, if $V_{Src} \not\subset V_{jSrc}$
 $E_{Dst2Src} = \{e \mid e = (\text{name}, \text{Type}(pe), v_{Src}, v_{dst}), v_{Src} \in V_{jSrc} \wedge v_{dst} \in V_{Dst}\}$
 $eb = (pe, E_{Dst2Src})$
 $vb = (\text{Src}(pe), V_{iSrc})$
 add eb to EB where $m = (VB, EB)$
 add vb to VB where $m = (VB, EB)$
 MatchDynamic(pg, hg, m)
 delete eb from EB where $m = (VB, EB)$
 delete vd from VB where $m = (VB, EB)$

APPENDIX D.

FORMAL SEMANTICS OF GREAT

A formal specification of the GReAT execution semantics is described in this Chapter. The Object-Z notation [101] has been used for the specification. The specification starts with the definition of a graph.



Vertices and edges both have a type associated with them. These types must conform to the respective metamodels of the graphs. Both host graphs and pattern graphs are defined by the same data structure. The additional attributes of the pattern graph, like actions are captured separately using maps. The *MATCH* class is a data structure that associates pattern graph elements with host graph elements. (The host graph is the graph in which we search for a match.) It contains a partial function from pattern vertices to host vertices and another partial function that maps pattern edges to host edges.

MATCH

$$\begin{array}{l} \text{hostGraph} : \text{GRAPH} \\ \text{patternGraph} : \text{GRAPH} \\ \text{vertexBinding} : [\text{VERTEX} \leftrightarrow \text{VERTEX}] \\ \text{edgeBinding} : [\text{EDGE} \leftrightarrow \text{EDGE}] \end{array}$$
$$\begin{array}{l} \forall (hv, pv) \in \text{vertexBinding} \bullet \\ \quad hv \in \text{hostGraph.vertices} \wedge \\ \quad \quad pv \in \text{patternGraph.vertices} \\ \forall (he, pe) \in \text{edgeBinding} \bullet \\ \quad he \in \text{hostGraph.edges} \wedge \\ \quad \quad pe \in \text{patternGraph.edges} \wedge \\ \quad \quad \exists (hvs, pvs), (hvd, pvd) \in \text{vertexBindings} \bullet \\ \quad \quad \quad he.src = hvs \wedge pe.src = pvs \\ \quad \quad \quad he.dst = hvd \wedge pe.dst = pvd \end{array}$$

Apart from the pattern graph, a rule also contains ports that allow it to interface with other rules. A port is simply used to connect with another rule. A non-empty set of ports form an interface. Each rule must contain an input and an output interface. The interface is used to pass along host graph elements. These elements are mapped to the ports of an interface to form a packet. A *PACKET* contains a partial function that maps ports to host vertices.

PORT

$$\text{name} : \text{NAME}$$

INTERFACE

$$\text{ports} : \mathbb{P} \text{PORT}$$

PACKET

$$\text{p2vMAP} : (\text{PORT} \mapsto \text{VERTEX})$$

The base class for all elements in the GReAT language that describes some operation on the graph is called *UNIT*. A *UNIT* consists of (1) a (reference to the) host

graph, (2) an input interface (3) an output interface, (4) a set of input packets, and (4) a set of output packets. *UNIT* is then specialized into *PRIMITIVE_UNIT* and *COMPOUND_UNIT*. *PRIMITIVE_UNIT* is specialized into *RULE* and *CASE*. These classes form the atomic building blocks of the GReAT language. The *RULE* performs an elementary transformation operation while *CASE* is used to check for matches (alternatives).

UNIT

```

hostGraph : GRAPH
inputInterface : INTERFACE
outputInterface : INTERFACE
inPackets :  $\mathbb{P}$  PACKET
outPackets :  $\mathbb{P}$  PACKET

```

PRIMITIVE_UNIT

```

UNIT
patternGraph : GRAPH
inBindings : [PORT  $\leftrightarrow$  VERTEX]
outBindings : [PORT  $\leftrightarrow$  VERTEX]
matches :  $\mathbb{P}$  MATCH
guard : OCL_EXPRESSION

```

```

 $\forall port \in dom(inBinding) \bullet port \in inputInterface.ports$ 
 $\forall port \in dom(outBinding) \bullet port \in outputInterface.ports$ 
 $\forall vertex \in range(outBinding) \vee range(inBinding) \bullet$ 
    $vertex \in patternGraph.vertices$ 

```

MakeInitialPartialMatch

```

initialPartialMatch! : MATCH

```

```

inPackets' = inPackets - inPacket
 $\forall (p, hv) \in inPacket.p2vMap \bullet \exists (p, pv) \in inBinding \bullet$ 
    $(hv, pv) \in initialPartialMatch.vertexBinding$ 

```

PRIMITIVE_UNIT_contd...

PatternMatcher

hostGraph? : GRAPH
patternGraph? : GRAPH
initialPartialMatch? : MATCH

$\forall m \in matches \bullet m \supseteq initialPartialMatch$
 $\forall v \in patternGraph.vertices \bullet$
 $\quad \exists(hv, pv) \in m.vertexBinding \wedge hv = v$
 $\forall e \in patternGraph.edges \bullet$
 $\quad \exists(he, pe) \in m.edgeBinding \wedge he = e$
 $\forall m1, m2 \in matches \bullet m1 \neq m2$

EvaluateGuard

guard? : OCL_EXPRESSION

$\forall match \in matches \bullet$

Evaluate guard expression on match. If evaluation results false then remove match from matches.

PackageResult

match? : MATCH
outPacket! : PACKET

$\forall p \in outputInterface.ports \bullet$
 $\quad \exists(p, pv) \in outBinding \wedge \exists(hv, pv) \in match$
 $\quad outPacket' = outPacket \oplus p \mapsto hv$
 $\quad outPackets' = outPackets \cup outPacket$

$MakeInitialPartialMatches \hat{=} MakeInitialPartialMatches_{\S}$

$MakeInitialPartialMatches$

\parallel

$[inPackets = \{\}]$

$PackageResults \hat{=} PackageResult_{\S} PackageResults$

\parallel

$[matches = \{\}]$

PRIMITIVE_UNIT contains a pattern graph, binding of input ports to pattern elements and binding of pattern elements to output ports. It also contains many operations that are used by RULE and CASE. The most important operation is PatternMatcher. This

operation takes as input a partial match of the pattern on the host graph and generates the set of all possible complete matches between the pattern and the host graph. This matcher algorithm implements the core activity performed during the execution of GReAT programs. The other operations include: *MakeInitialPartialMatch*, that takes a single input packet and converts it into a partial match using the input binding information, and *EvaluateGuard* that is used to evaluate an OCL expression on the matches returned by the matcher. All matches that fail the guard are discarded. For the sake of brevity the *EvaluateGuard* function is described in English.

<p><i>CASE</i></p> <hr/> <p><i>PRIMITIVE_UNIT</i></p> <p><i>Execute</i> $\hat{=}$ <i>MakeInitialPartialMatches</i> § <i>PatternMatcher</i> § <i>EvaluateGuard</i> § <i>PackageResults</i></p> <p style="text-align: center;">⌈</p> <p style="text-align: center;">[<i>inputPackets</i> = {}]</p>

A *CASE* is the simplest of all GReAT components. The *Execute* function of the case takes each input packet and calls the pattern matcher. The matches returned by the pattern matcher are then filtered using the guard expression. All successful matches are again packaged to form the output packets. The *CASE* is used only within a *TEST* component. *TEST* and *CASE* are used together, to form a conditional execution and branching construct.

The execution of a *RULE* is similar to that of a *CASE*. The exception is that in a *RULE*, after the matches are filtered using the guard, the matches are used to perform actions on the host graph. These actions can create and/or delete vertices and edges. After these actions are performed, the attribute mapping specification is used by *PerformAttributeMapping* operation to fill in and/or modify the attributes of graph

vertices and edges. For the sake of brevity, PerformAttributeMapping is described in English.

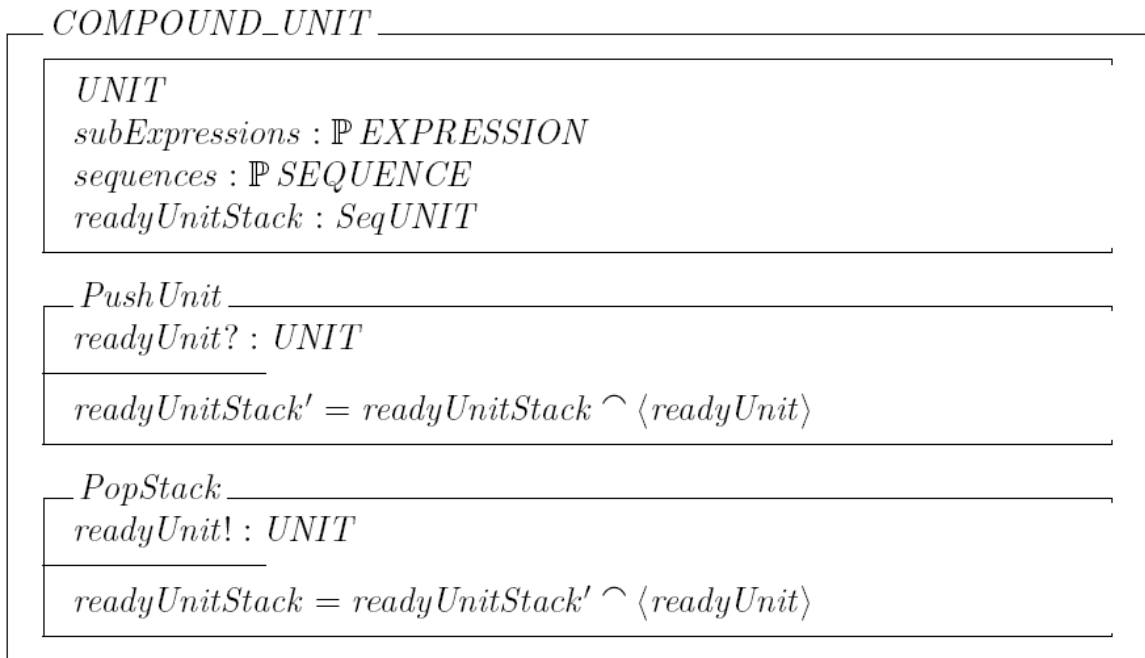
<p><i>RULE</i></p> <hr/> <p><i>PRIMITIVE_UNIT</i></p> <p><i>ACTION</i> == {<i>bind</i>, <i>create</i>, <i>delete</i>}</p> <hr/> <p><i>vertexAction</i> : [<i>VERTEX</i> \leftrightarrow <i>ACTION</i>]</p> <p><i>edgeAction</i> : [<i>EDGE</i> \leftrightarrow <i>ACTION</i>]</p> <p><i>attributeMapping</i> : \mathbb{P} <i>ASSIGNMENT_STATEMENTS</i></p> <hr/> <p><i>PerformAction</i></p> <hr/> <p>$\forall match \in matches \bullet$</p> <p style="padding-left: 20px;">$\forall (v, a) \in vertexAction \bullet ACTION = create$</p> <p style="padding-left: 40px;">$hostGraph.vertices' = hostGraph.vertices \cup new_v : VERTEX$</p> <p style="padding-left: 40px;">$\wedge new_v.name = v.name \wedge new_v.type = v.type$</p> <p style="padding-left: 20px;">$\forall (e, a) \in edgeAction \bullet ACTION = create$</p> <p style="padding-left: 40px;">$hostGraph.edges' = hostGraph.edges \cup new_e : EDGE \wedge$</p> <p style="padding-left: 40px;">$new_e.name = e.name \wedge new_e.type = e.type$</p> <p style="padding-left: 20px;">$\forall (v, a) \in vertexAction \bullet ACTION = delete$</p> <p style="padding-left: 40px;">$hostGraph.vertices' = hostGraph.vertices - v$</p> <p style="padding-left: 20px;">$\forall (e, a) \in edgeAction \bullet ACTION = delete$</p> <p style="padding-left: 40px;">$hostGraph.edges' = hostGraph.edges - e$</p> <hr/> <p><i>PerformAttributeMapping</i></p> <hr/> <p>$\forall match \in matches \bullet$</p> <p>Apply attribute matching statements on the match.</p> <hr/> <p><i>Execute</i> $\hat{=}$ <i>MakeInitialPartialMatches</i> ; <i>PatternMatcher</i> ;</p> <p style="padding-left: 40px;"><i>EvaluateGuard</i> ; <i>PerformAction</i> ;</p> <p style="padding-left: 40px;"><i>PerformAttributeMapping</i> ; <i>PackageResults</i></p>

Sequential execution of expressions is expressed using the SEQUENCE class. This class maps ports of one UNIT to ports of another UNIT. SEQUENCE is usually used to map from the output interface of one UNIT to the input interface of another

UNIT. However, it is seen that in compound units SEQUENCE is also used to map the input interface of the compound unit to the input interface of contained units.

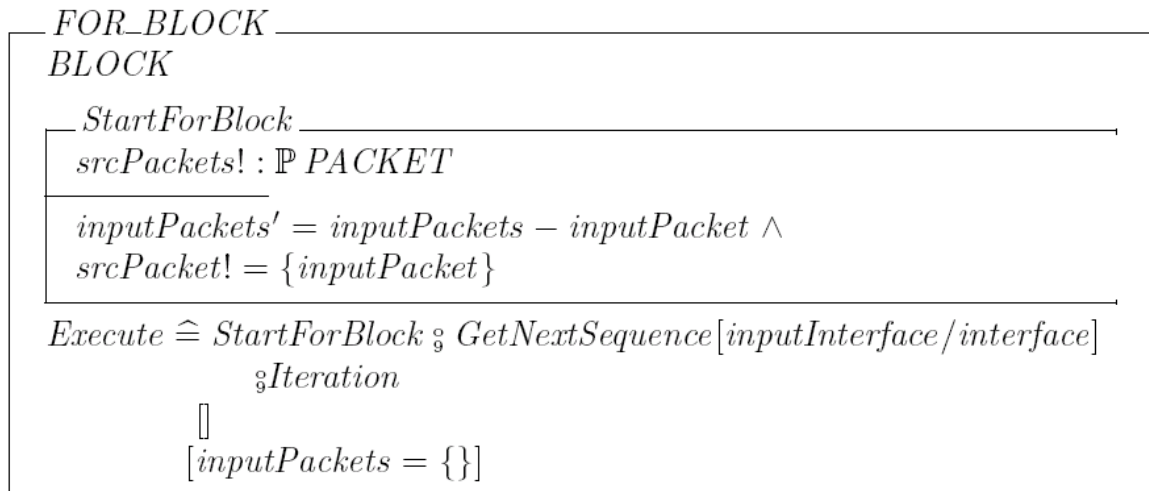


TEST is a UNIT that provides the language with a conditional execution and branching construct. A TEST contains an ordered sequence of CASE-es. The execution semantics of the TEST is that each CASE within a TEST is executed in order, starting from the first case in the sequence. COMPOUND UNIT is the base class of the two compound objects in GReAT: (1) BLOCK and (2) FOR_BLOCK. These blocks are useful for encapsulating complex rule sequences. The only difference between a BLOCK and FOR_BLOCK is in their execution semantics. The compound expressions use a stack machine semantics and thus have a ready UnitStack with push and pop operations.



The BLOCK is the simplest compound unit. It encapsulates a set of units along with their sequencing. The execution of the block starts with the StartBlock function that finds all the units that have a sequence from the input interface of the BLOCK. All these units are added to the readyUnitStack along with a copy of the input packets set of the BLOCK.

and add these to the stack. This process is repeated until the readyUnitStack is empty. Whenever a unit that has executed is connected to the output interface of the BLOCK, the outputs are copied to the output of the BLOCK. The FOR BLOCK is similar to the BLOCK with a subtle difference. The execution of the FOR BLOCK starts the unit execution stack with only the first input packet. When the stack is empty the process is repeated with the next packet until all packets are exhausted. The FOR BLOCK provides a depth first execution of all the contained units while the BLOCK provides a breadth first execution.



CONFIGURATION ASPECT OF UMT

Information that is required to run the transformations such as the starting rule of the transformation, inputs to and outputs from the transformation, the files involved etc. are captured by the configuration aspect of the UML Model Transformer (UMT) paradigm.

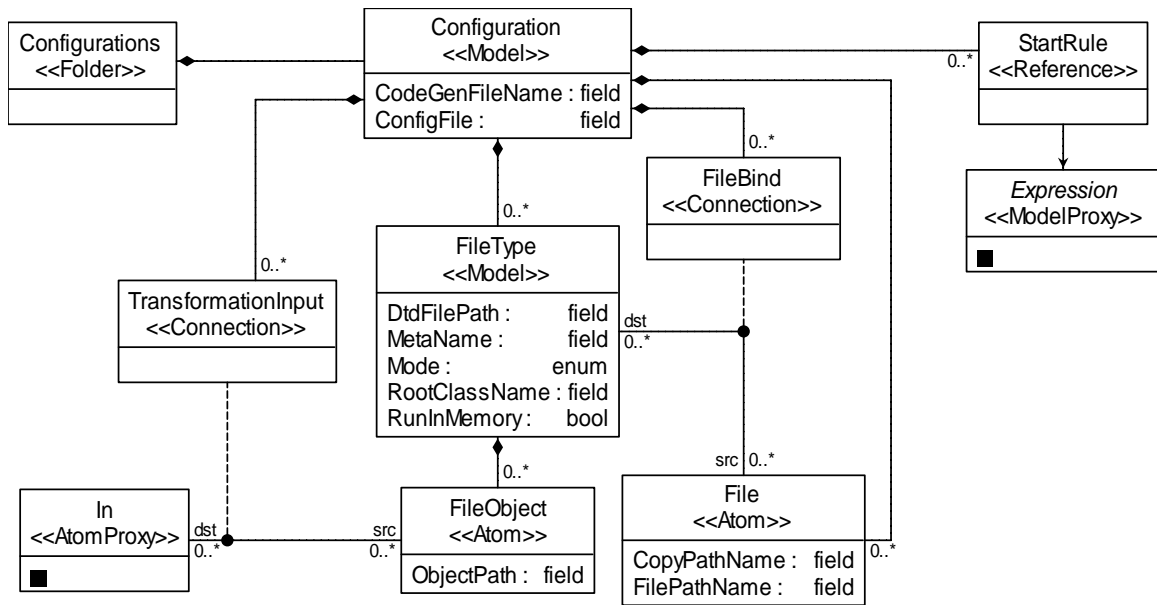


Figure 64 Metamodel of the configuration aspect of UMT

Figure 64 shows the metamodel of the configuration aspect. A configuration contains a *StartRule* that can refer to any *Expression*. The configuration also contains *FileType* objects that define the type of the file the particular input will belong to. *FileType* instances will contain the paradigm, root object name, the dtd/xsd path, the file operation mode and other information that deals with the file handling. *FileTypes*

contain *FileObjects* which are place holders defining the particular object in the file that will be provided as input to the start rule. *TransformationInput* is an association that associates the *FileObjects* with the input ports of the *StartRule*. Entities called *File* provide the names of the default files to be used to run the transformations and are associated with their *FileTypes* using the *FileBind* association.

APPENDIX F.

THE SIMULINK/STATEFLOW TO HSIF TRANSLATION ALGORITHM

This algorithm was developed by Dr. Gyula Simon.

Definition 1 The flat Stateflow state machine contains the set of state $S = \{s_1, s_2, \dots, s_N\}$, s_1 being the initial state. The set of transitions is $T \subseteq S \times S$ where $t_{i,j} \in T$ is a transition from s_i to s_j . The corresponding transition condition is denoted by $w_{i,j}$.

Definition 2 An output variable in the Stateflow diagram is called a switching signal if it is connected to a Control Input of a Switch block in the Simulink diagram. The set of switching signals in the state machine is $Q = \{q_1, q_2, q_3, \dots, q_M\}$. The value of the switching signal q in state s is $value(q, s)$.

Definition 3 The *switch value* of a switching signal q in state s is the following:

$$switchvalue(q, s) = \begin{cases} 1 & \text{if } value(q, s) \geq threshold(b) \\ 0 & \text{otherwise} \end{cases}$$

where b is the unique Switch block connected to q .

Definition 4 For a switching signal q and state s_i , $defined(q, s_i) = true$ if either of the following conditions hold:

- q is explicitly set in s_i , or

- there exist a switch value u , such that for all j for which $t_{j,i} \in T$ it is true that $defined(q, s_j)$ and $switchvalue(q, s_j) = u$.

Definition 5 The *rank* of state s is the number of switching signals that are defined in s . The *defect* of s is defined as $defect(s) = M - rank(s)$.

Definition 6 The sequence of undefined switching signals in s_i is defined as

$$U_i = \left\langle q_{k_1}, q_{k_2}, q_{k_3}, \dots, q_{k_{defect(s_i)}} \right\rangle, \quad \text{where } defined(q_{k_l}, s_i) = false \quad \text{for all } l = 1, 2, \dots, defect(s_i), \text{ and } k_1 < k_2 < \dots < k_{defect(s_i)}.$$

The algorithm consists of the following steps.

Step 1. Each state s_i is split into $D = 2^{defect(s_i)}$ locations. The set of locations generated from s_i is $\sum_i = \{\sigma_{i,1}, \sigma_{i,2}, \dots, \sigma_{i,D}\}$.

Definition 7 The switch code of location $\sigma_{i,j}$ is a binary sequence of length M , denoted by $C_{i,j} = \langle b_{i,j,1}, b_{i,j,2}, \dots, b_{i,j,M} \rangle$. The binary values are defined as follows:

$$b_{i,j,k} = \begin{cases} switchvalue(q_k, s_i) & \text{if } q_k \in U_i \\ bit(j-1, n) & \text{if } q_k = q_{k_n}, \text{ where } U_i = \langle q_{k_1}, \dots, q_{k_{defect(s_i)}} \rangle \end{cases}$$

The function $bit(x, y)$ defines the y^{th} bit of the binary representation of x , the 1st bit being the least significant bit.

Definition 8 The coloring is defined on the elements of the switch code. The binary values of the code are either black or red, as follows:

$$color(b_{i,j,k}) = \begin{cases} red & \text{if } q_k \in U_i \\ black & \text{if } q_k \notin U_i \end{cases}$$

Step 2. The locations are coded and colored according to Definition 7 and Definition 8.

Step 3. Create a transition $\tau_{i,j,n,m}$ between $\sigma_{i,n}$ and $\sigma_{j,m}$ if $t_{ij} \in T$, and there is no k such that $b_{i,n,k} \neq b_{j,m,k}$ and $color(b_{j,m,k}) = red$. The transition guard for this transition is the predicate w_{ij} .

Definition 9 The set of all transitions in the HSIF description is denoted by Φ .

Definition 10 The Simulink diagram containing M Switch blocks describes the reconfigurable dynamic system χ . The dynamic system with a particular setting of the switches with switch values x_1, x_2, \dots, x_M is denoted by $\chi(x_1, x_2, \dots, x_M)$.

Step 4. For each state s_i copy the algebraic equations defined in the state to locations $\sigma_{i,j}$, for all $j = 1, 2, 3, \dots, 2^{defect(s_i)}$. For each location $\sigma_{i,j}$ generate the additional algebraic and differential equations of the system $\chi(C_{i,j})$.

Step 5. Choose $\sigma_{1,1}$ to be the initial location.

Step 6. Add the following invariants to location $\sigma_{i,j}$:

- switching signal values from the entry action of s_i , and
- $\neg(\bigvee_m w_{i,m})$ for all indices m for which there exist n such that $\tau_{i,m,j,n} \in \Phi$. The operations \neg and \bigvee are the logical not and or operations, respectively.

Definition 11 The *location dependency graph* is a directed graph on the set $\sum_1 \cup \sum_2 \cup \dots \cup \sum_N$ with edges Φ . A location σ is *unreachable* if there is no directed path in the location dependency graph from $\sigma_{1,1}$ to σ .

Step 7. Prune all unreachable locations from the HSIF description. Also delete the transitions connected to unreachable locations.

REFERENCES

- [1] J. Sztipanovits, and G. Karsai, "Model-Integrated Computing", Computer, Apr. 1997, pp. 110-112
- [2] "The Model Driven Architecture", OMG, Needham, MA, 2002, URL = <http://www.omg.org/mda/>.
- [3] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.
- [4] "Request For Proposal: MOF 2.0 Query/Views/Transformations", OMG Document: ad/2002-04-10, 2002, OMG, Needham, MA.
- [5] Levine, J., T. Mason and D. Brown, "lex & yacc", O'Reilly, 1992, 2nd edition.
- [6] Agrawal A., Karsai G., Ledeczi A., "An End-to-End Domain-Driven Development Framework", Domain-driven development track, 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, California, October 26, 2003.
- [7] Bruno G., "Model Based Software Engineering", Chapman & Hall, 1995.
- [8] "Model of Computation", Dictionary of Algorithms and Data Structure, National Institute of Standards and Technology, URL = <http://www.nist.gov/dads/HTML/modelofcompu.html>.
- [9] "Finite State Machine", Dictionary of Algorithms and Data Structure, National Institute of Standards and Technology, URL = <http://www.nist.gov/dads/HTML/finiteStateMachine.html>.
- [10] K. L. McMillan, "Symbolic Model Checking: an approach to the state explosion problem", CMU Tech Rpt. CMU-CS-92-131.
- [11] "Turing Machine", The Stanford Encyclopedia of Philosophy (Summer 2003 Edition), (ed.), URL = <http://plato.stanford.edu/archives/sum2003/entries/turing-machine/>.
- [12] "The Church-Turing Thesis", The Stanford Encyclopedia of Philosophy (E. Zalta, Edition), (ed), URL = <http://plato.stanford.edu/entries/church-turing/>.
- [13] E. A. Lee, <http://ptolemy.eecs.berkeley.edu/~eal/ee290n/glossary.html>, EE290N: Advanced Topics in System Theory, Fall, 1996.
- [14] A. Ledeczi, et al., "Composing Domain-Specific Design Environments", Computer, Nov. 2001, pp. 44-51.

- [15] J. D. Lara , H. Vangheluwe, “Using AToM3 as a Meta-CASE Tool”, Proceedings of the 4th International Conference on Enterprise Information Systems ICEIS'2002 , 642-649, Ciudad Real, Spain, April 2002.
- [16] “Dome Guide”, Honeywell, Inc. Morris Township, N.J, 1999.
- [17] Kim Mason, “Moses Formalism Creation – Tutorial”, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, CH-8092, Switzerland, February 9, 2000.
- [18] L. A. Cortes, P. Eles, and Z. Peng, “A Survey on Hardware/Software Codesign Representation Models”, *SAVE Project Report*, Dept. of Computer and Information Science, Linköping University, Sweden, June 1999.
- [19] A. Jerraya and K. O’Brien, “SOLAR: An Intermediate Format for System-Level Modeling and Synthesis,”, *Codesign: Computer-Aided Software/Hardware Engineering*, J. Rozenblit and K. Buchenrieder, Eds. Piscataway, NJ, IEEE Press, 1995, pp. 145-175.
- [20] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vicentelli, “A Formal Specification Model for Hardware/Software Codesign,” *Technical Report UCB/ERL M93/48*, Dept. EECS, University of California, Berkeley, June 1993.
- [21] D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming*, vol. 8, pp. 231-274, June 1987.
- [22] C. G. Cassandras, “Discrete Event Systems: Modeling and Performance Analysis”, *Irwin Publications*, Boston, MA, 1993.
- [23] E. A. Lee, “Modeling Concurrent Real-Time Processes using Discrete Events,” *Technical Report UCB/ERL M98/7*, Dept. EECS, University of California, Berkeley, March 1998.
- [24] J. Peterson, “Petri Net Theory and the Modeling of Systems”, *Prentice-Hall, Englewood Cliffs, NJ*, 1981.
- [25] G. Dittrich, “Modeling of Complex Systems Using Hierarchical Petri Nets,” *Codesign: Computer-Aided Software/Hardware Engineering*, J. Rozenblit and K. Buchenrieder, Eds. Piscataway, NJ: IEEE Press, 1995, pp. 128-144.
- [26] T. De Marco, “Structured Analysis and System Specification”, Prentice-Hall, Englewood Cliffs, NJ, 1978.
- [27] C. P. Gane and T. Sarson, “Structured System Analysis: Tools and Techniques”, Prentice-Hall International, Englewood Cliffs, NJ, 1979.

- [28] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *Transactions on Computers*, C36 (1): 24-35, January 1987.
- [29] Katsuhiko Ogata, "Modern Control Engineering", 4th edition. Prentice Hall, 2001.
- [30] R. J. Mayers, et al., "Information Integration For Concurrent Engineering (Iice) Idef3 Process Description Capture Method Report", Human Resources Directorate Logistics Research Division, Knowledge Based Systems, Incorporated, Texas 77840-2335, September 1995.
- [31] A. Kalavade, Edward A. Lee, "Design Methodology Management For System-Level Design", Ptolemy Miniconference, March 10, 1995.
- [32] A. Kalavade, E. A. Lee, "A Global Criticality/Local Phase driven Algorithm for the Constrained Hardware/Software Partitioning Problem", *Proc. of Codes/CASHE'94, Third Intl. Workshop on Hardware/Software Codesign*, pp. 42-48, Sept. 22-24, 1994.
- [33] Edward A. Lee, "Overview of the Ptolemy Project", *Technical Memorandum UCB/ERL M01/11* March 6, 2001.
- [34] P. P. Chen. "The Entity-Relationship Model". *ACM Trans. on Database Systems (TODS)*, 1:9-36, 1976.
- [35] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.
- [36] M. Fowler, "UML Distilled Second Edition", Addison Wesley Longman, Inc., 200.
- [37] J. Gray, G. Karsai, "An Examination of DSLs for Concisely Representing Model Traversals and Transformations", 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9, p. 325a, January 06 - 09, 2003.
- [38] "Simulink Reference", The Mathworks, Inc., July 2002.
- [39] ActiveHDL, <http://www.aldec.com/ActiveHDL/>, Aldec Inc., Henderson, NV 89074.
- [40] M. E. Lesk, "LEX---a lexical analyzer generator", CSTR 39, Bell Laboratories, Murray Hill, NJ, 1975.
- [41] Johnson S.C., "Yacc: Yet Another Compiler-Compiler", Bell Laboratories, Murray Hill, NJ, 1978.
- [42] K. Czarnecki, U. Eisenecker, "Generative Programming: Methods, Techniques, and Applications", Addison-Wesley, 1999.
- [43] J. Neighbors, "Software Construction Using Components", Ph.D. Thesis, ICS-TR-160, University of California at Irvine, 1980.

- [44] J. Neighbors, “Draco 1.2 Users Manual”, University of California at Irvine, 1983.
- [45] Don S. Batory, Jacob Neal Sarvela, Axel Rauschmayer, “Scaling Step-Wise Refinement”, International Conference on Software Engineering, pp 187-197, 2003.
- [46] The Moses Project, Computer Engineering and Communications Laboratory, ETH Zurich URL = <http://www.tik.ee.ethz.ch/~moses/>
- [47] R. Essar, J. Janneck and M. Naedele, “The Moses Tool Suite - A Tutorial”, Version 1.2, Computer, Engineering and Networks Laboratory, ETH Zurich, 2001.
- [48] J. Janneck, “Graph-type definition language (GTDL)—specification”, Technical report, Computer, Engineering and Networks Laboratory, ETH Zurich, 2000.
- [49] J. Lara , H. Vangheluwe, “Using AToM as a Meta CASE Tool”, 4th International Conference on Enterprise Information Systems, Universidad de Castilla-La Mancha, Ciudad Real (Spain), 3-6, April 2002.
- [50] J. Lara, H. Vangheluwe, “Computer Aided Multi-Paradigm Modeling to Process Petri-Nets and Statecharts”, 1st International Conference on Graph Transformation, Barcelona (Spain), 7-12, October 2002.
- [51] S. Kent, O. Patrascoiu, “Kent Modelling Framework Version – Tutorial”, Computing Laboratory, University of Kent, Canterbury, UK, Draft, December 2002.
- [52] “ABC To Metacase Technology”, White Paper, MetaCase Consulting, Finland, August, 2000.
- [53] “Domain-Specific Modelling: 10 Times Faster Than UML”, White Paper, MetaCase Consulting, Finland, January, 2001.
- [54] Grzegorz Rozenberg, “Handbook of Graph Grammars and Computing by Graph Transformation”, World Scientific Publishing Co. Pte. Ltd., 1997.
- [55] M. Nagl, “Formal Languages of Labeled Graphs”, Computing 16 (1976), 113-137.
- [56] M. Kaul, “Practical applications of precedence graph grammars”, Graph Grammars and their application to Computer Science, Lecture Notes in Computer Science 291, Springer-Verlag, Berlin, 1987.
- [57] G. Rozenberg, E. Welzl, “Graph Theoretic closure properties of the family of boundry NLC graph languages”, Acta Informatica 23, 289-309, 1986.
- [58] R. Schuster, “Graphgrammatiken und Grapheinbettungen”, Algorithmen und Komplexitat, Technical Report MIP-8711, Universitat Passau, 1987.
- [59] Annegret Habel, “Hyperedge Replacement: Grammars and Languages”, volume 643 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1992.

- [60] Annegret Habel, "Hypergraph Grammars: Transformational and algorithmic aspects", *Journal of Information Processing and Cybernetics EIK*, 28:241-277, 1992.
- [61] Michel Bauderon and Bruno Courcelle, "Graph expressions and graph rewriting", *Mathematical Systems Theory*, 20:83-127, 1987.
- [62] R. J. Parikh, "On context-free languages", *Journal of ACM*, 13:570-581, 1966.
- [63] H. Erig, M. Pfender, and H. J. Schneider, "Graph Grammars: an algebraic approach", In *Proceedings IEEE Conf. on Automata and Switching Theory*, pages 167-180, 1973.
- [64] M. Lowe, "Algebraic approach to single-pushout graph transformation", *Theoretical Computer Science*, 109:181-224, 1993.
- [65] Blostein D., Schürr A., "Computing with Graphs and Graph Rewriting", Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany.
- [66] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *Communications of ACM*, 18:453-457, 1975.
- [67] G. Nelson, "A Generalization of Dijkstra's Calculus", *ACM transactions on Programming Languages and Systems*, Vol. 11, No. 4, pp-517-561, 1989.
- [68] A. Schürr, "PROGRES for Beginners", Technical Report, Lehrstuhl für Informatik III, RWTH Aachen, Germany.
- [69] H. Gottler, "Attributed graph grammars for graphics", H. Ehrig, M. Nagl, and G. Rosenberg, editors, *Graph Grammars and their Application to Computer Science*, LNCS 153, pages 130-142, Springer-Verlag, 1982.
- [70] H. Göttler, "Diagram Editors = Graphs + Attributes + Graph Grammars," *International Journal of Man-Machine Studies*, Vol 37, No 4, Oct. 1992, pp. 481-502.
- [71] C. Ermel, T. Schultzke, "The AGG Environment: A Short Manual", TU Berlin.
- [72] J. Loyall and S. Kaplan, "Visual Concurrent Programming with Delta-Grammars," *Journal of Visual Languages and Computing*, Vol 3, 1992, pp. 107-133.
- [73] D. Blostein, H. Fahmy, and A. Grbavec, "Practical Use of Graph Rewriting", 5th Workshop on Graph Grammars and Their Application To Computer Science, *Lecture Notes in Computer Science*, Heidelberg, 1995.
- [74] U. Assmann, "How to Uniformly specify Program Analysis and Transformation", *Proceedings of the 6 International Conference on Compiler Construction (CC) '96*, LNCS 1060, Springer, 1996.

- [75] A. Maggiolo-Schettini, A. Peron, "A Graph Rewriting Framework for Statecharts Semantics", Proc.\ 5th Int.\ Workshop on Graph Grammars and their Application to Computer Science, 1996.
- [76] A. Radermacher, "Support for Design Patterns through Graph Transformation Tools", Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, The Netherlands, Sep. 1999.
- [77] A. Bredendfeld, R. Camposano, "Tool integration and construction using generated graph-based design representations", Proceedings of the 32nd ACM/IEEE conference on Design automation conference, p.94-99, June 12-16, 1995, San Francisco, CA.
- [78] H. Fahmy, B. Blostein, "A Graph Grammar for Recognition of Music Notation", Machine Vision and Applications, Vol. 6, No. 2 (1993), 83-99.
- [79] G. Engels, H. Ehrig, G. Rozenberg (eds.), "Special Issue on Graph Transformation Systems", Fundamenta Informaticae, Vol. 26, No. 3/4 (1996), No. 1/2, IOS Press (1995).
- [80] G.Schmidt, R. Berghammer (eds.), "Proc. Int. Workshop on Graph-Theoretic Concepts in Computer Science", (WG '91), LNCS 570, Springer Verlag (1991).
- [81] H.Ehrig, M. Pfender, H. J. Schneider, "Graph-grammars: an algebraic approach", Proceedings IEEE Conference on Automata and Switching Theory, pages 167-180 (1973).
- [82] G. Viehstaedt, M. Minas, "Generating editors for direct manipulation of diagrams", 5th International Conference on Human-Computer Interaction, Moscow, Russia, pages 17-25. Springer-Verlag, July 1995.
- [83] Bardohl,R., Ermel,C., and Weinhold,I., "GenGED - A visual definition tool for visual modeling environments", Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03), pages 407-414, Sept./Oct., 2003, Charlottesville, Virginia, USA.
- [84] D. Varro, G. Varro and A. Pataricza, "Designing the Automatic Transformation of Visual Languages", volume 44, Elsevier, pages 205–227, Science of Computer Programming, 2002.
- [85] Vizhanyo A., Agrawal A., Shi F., "Towards Generation of High-performance Transformations", Generative Programming and Component Engineering, Vancouver, Canada, October 24, 2004.
- [86] Agrawal A., Simon G., Karsai G., "Semantic Translation of Simulink/Stateflow models to Hybrid Automata using Graph Transformations", International Workshop on Graph Transformation and Visual Modeling Techniques, Barcelona, Spain,

March 27, 2004, To be published in Electronic Notes in Theoretical Computer Science.

- [87] Object Management Group, Object Constraint Language Specification, OMG Document formal/01-9-77. September 2001.
- [88] A. Bakay, “The UDM Framework,” <http://www.isis.vanderbilt.edu/Projects/mobies/>.
- [89] Magyari E., Bakay A., Lang A., Paka T., Vizhanyo A., Agrawal A., Karsai G.: “UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages”, The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003, Anaheim, California, October 26, 2003.
- [90] J. McCarthy “Recursive functions of symbolic expressions and their computation by machine – I”, Communications of the ACM, 3(1), 184-195, 1960.
- [91] Uwe Assmann, “Aspect Weaving by Graph Rewriting”, Generative Component-based Software Engineering (GCSE), p. 24-36, Oct 1999.
- [92] G. Karsai, S. Padalkar, H. Franke, J. Sztipanovits, ”A Practical Method For Creating Plant Diagnostics Applications”, Integrated Computer-Aided Engineering, 3, 4, pp. 291-304, 1996.
- [93] E. Long, A. Misra, J. Sztipanovits, “Increasing Productivity at Saturn”, IEEE Computer Magazine, August 1998.
- [94] AGG, <http://tfs.cs.tu-berlin.de/agg/>.
- [95] H. Kreowski, S. Kuske: “Graph Transformation Units and Modules,” in H. Ehrig, G. Engels, H. Kreowski, G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, pages 607-638. World Scientific, Singapore, 1999.
- [96] Karsai G., Agrawal A., Shi F., Sprinkle J., “On the Use of Graph Transformations for the Formal Specification of Model Interpreters”, Journal of Universal Computer Science, Special issue on Formal Specification of CBS, 2003.
- [97] J. Gray, G. Karsai, “An Examination of DSLs for Concisely Representing Model Traversals and Transformations”, 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9, p. 325a, January 06 - 09, 2003.
- [98] The Hybrid System Interchange Format, for details see <http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp>
- [99] T. A. Heinzinger, “The Theory of Hybrid Automata”, In Proc. Of IEEE Symposium on Logic in Computer Science, IEEE press, pp 278-292, 1996.

- [100] Hylands, C., Lee, E., Liu, J., Liu, X., Neuendorffer, S., Zheng, H., “HyVisual: A Hybrid System Visual Modeler,” Technical Memorandum UCB/ERL M03/1, University of California, Berkeley, CA 94720, January 28, 2003.
- [101] Roger Duke, Gordon Rose and Graeme Smith, “Object-Z: a Specification Language Advocated for the Description of Standards”; TR 94-95, December 1994, Software Verification Research Centre, Department Of Computer Science, The University Of Queensland, Queensland 4072, Australia.
- [102] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine – I”, Communications of the ACM, 3(1), 184-195, 1960.