Model Based Software Engineering, Graph Grammars and Graph Transformations

Area Paper

Aditya Agrawal

March 2004

Committee:

Gabor Karsai, Chair
Janos Sztipanovits
Doug Schmidt
Jeremy Spinrad
Mark Ellingham
Gautam Biswas

Table of Contents

Model Based Software Engineering and Graph Grammars and Transformations 1
1. Introduction 4
2. Model Based Software Engineering 5
    2.1. Model Classification 5
        2.1.1. Low-level and High-level models 6
        2.1.2. Models Of Computation 6
            2.1.2.1. Finite State Machine (FSM) 6
            2.1.2.2. Turing Machine 7
            2.1.2.3. Discrete-Event Systems 8
            2.1.2.4. Petri Nets 8
            2.1.2.5. Data Flow Graph 9
        2.1.3. High-level models 10
            2.1.3.1. Unified Modeling Language (UML) 10
        2.1.4. Low-level Vs High-level 11
    2.2. Domain Specific Modeling 12
        2.2.1. IDEF3 - Process Flow and Object State Description Capture Method 12
        2.2.2. Ptolemy II – A polymorphic design environment 13
        2.2.3. Domain Specific Vs Domain Independent 14
3. Generative and Model Based Solutions 16
    3.1. Generative Programming (GP) 16
        3.1.1. Draco 17
        3.1.2. GenVoca 17
        3.1.3. Summary 18
    3.2. Model Integrated Computing (MIC) 18
        3.2.1. The DOmain Modeling Environment (DOME) 20
        3.2.2. Moses 20
        3.2.3. Atom^3 21
        3.2.4. Kent Modeling Framework 21
        3.2.5. MetaEdit+ 21
        3.2.6. Generic Modeling Environment (GME) 22
        3.2.7. Comparison of features 22
    3.3. Sumary of Model Based solutions 23
4. Graph Grammars and Transformations 25
    4.1. Node Replacement Graph Grammars 25
        4.1.1. Node Label Controlled (NLC) 25
        4.1.2. Neighborhood Controlled Embedding (NCE) 27
    4.2. Hyperedge Replacement Graph Grammars 29
    4.3. Algebraic Approach to Graph Transformation 30
        4.3.1. Double PushOut (DPO) 31
        4.3.2. Single PushOut (SPO) 32
    4.4. Programmed Graph Rewriting SystemS 32
        4.4.1. Logic-based Structure Replacement System 33
    4.5. Programmed Structure Replacement Systems 35

4.6. Sumary of Graph Grammars and Transformations 36
5. Graph Transformation Based Tools 37
5.1. Progres 37
5.2. AGG 37
5.3. Comparison of Features 37
5.4. Graph Transformations Critique 38
6. Proposal 40
6.1. Research Hypothesis 42
6.2. Research Methods 42
6.3. Completion Criteria 43
6.4. Preliminary Work and Results 43
7. References 45

1. 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 [6] attempted to bridge large semantic gaps between very-high-level semantic models and programming languages, and failed. 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 [36] 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.

For textual languages and compiler design there is a vast literature in textual grammars, parsers, parser generators and other formal methods to specify and implement textual languages and compilers. Theory and formal methods equivalent to that of textual grammars can be of tremendous help to the modeling community. Graph grammars and transformations have been studied for over the 30 years and have produced many theoretical results. However, these results haven't been applied to the development of methodologies or tools that facilitate the development of modeling languages.

The application of the theory of graph grammars towards the design of modeling languages seems to have much potential. This paper surveys both areas and explores potential topics for dissertation research.

This paper is organized as follows: Chapter 2 reviews Model Based Software Engineering. A survey of various modeling approaches is presented along with a critique. Chapter 3 investigates generative and model based solutions to solving software development problems in various domains. Generative methods such as Draco and GenVoca are surveyed. The chapter also looks at various meta configurable development tools used for the specification and implementation of domain specific modeling environments. A comparison of these tools is presented at the end of the chapter. Chapter 4 reviews the theoretical work on graph grammars and transformations and tries to see if it may be able to solve some problems identified in the meta-programmable tools. Chapter 5 contains a survey of some of the notable graph grammar and transformation based tools and evaluates the tools against a set of desired features. Finally in Chapter 6 the dissertation proposal is stated along with the goals, completion criteria and the current status of the research.

2. MODEL BASED SOFTWARE ENGINEERING

Software engineering is a discipline where a variety of challenges have to be met on a daily basis. These challenges are 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. 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 [6].

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

The main benefits of models are that they help abstract 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. These 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 suitable to express various aspects of software have been introduced and are being used in the community. Some example languages are dataflow and its variants [25][26][27], state machine and state charts [17][18][19][20], entity-relationship diagrams [32] and more recently object oriented modeling languages such as UML [33]. Several tools have been implemented that allow graphical specification of the structure and behavior of software using one or many of the formalisms mentioned above. These tools try to automate the process of simulation, verification and synthesis of the end system. [6]

In this chapter various modeling techniques are studied to find their merits and drawbacks. In order to study the advantages of modeling, modeling paradigms will be classified into different categories. The study may yield results that hold true for the entire classification of modeling language.

2.1. MODEL CLASSIFICATION

Models are an abstract representation of a system. They have various notations and are used for different purposes. Modeling languages can be classified based on various parameters. 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.

2.1.1. LOW-LEVEL AND HIGH-LEVEL MODELS

One criterion for classification is the level of abstraction: the level of detail captured by the models. On one hand low-level models, called Models of Computation (MoC) precisely describe how computation is done. On the other hand high-level models are used to specify design and intention. The next sub section will describe a few low-level and high-level modeling languages and compare them.

2.1.2. MODELS OF COMPUTATION

> *"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."*
>
> *[7]*

A 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 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.

2.1.2.1. Finite State Machine (FSM)

> *"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."*
>
> *[8]*

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 various weaknesses of the

classical FSM a number of extensions such as hierarchy and concurrency have been developed. A few such variants are discussed in brief [17].

SOLAR [18], 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) 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 broadcast and have zero propagation time. This model of computation is used as an intermediate representation that high-level languages can map to [17][19].

Statecharts by Harel [20] 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 and can be used to represent a wide range of systems from digital logic to communication protocols. FSMs have been widely studies and there are well established analysis methods and tools such as SMV [9]. On the other hand large problems with concurrency become very difficult to express due to the state space explosion. This prompted the introduction of a number of FSM variants that have introduced various abstractions to cover concurrency and hierarchy. SOLAR, CFSM or Statecharts can be considered as high-level modeling formalisms useful for the specification of large problems. Transformations can be written to map them to FSM which is a domain independent, platform independent MOC. By writing these transformations users not only benefit fro 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 isolating the implementation from intention.

2.1.2.2. Turing Machine

> *"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."*
>
> *[10]*

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 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 [11] and Turing completeness is used as a measure of the expressiveness of programming languages. In practice Turing machines are not used as programming for such a device is quite cumbersome. Instead high-level Turing complete programming languages such as C, C++ are used for programming needs.

2.1.2.3. Discrete-Event Systems

> *"Discrete-Event System is a timed system where for each tuple s of signals that satisfies the system, there is an order-preserving bijection from (a) the integers (for a two-sided DE system) or (b) the natural numbers (for a one sided DE system) to T(s), where T(s) is the set of tags in s."*
>
> *Edward. A. Lee [12]*

Systems having discrete states and driven by events over a period of time are referred to as Discrete-Event Systems [21]. These systems are asynchronous in nature and react to discrete events over time. An event is considered instantaneous, that is, the transition and action are performed in zero time. Discrete event systems are not restricted to finite number of states. The notion of event being tied to time is also a primary difference between FSMs and Discrete event systems.

Signals form the primary method of communication between tasks. They consist of a set of events over time. The events are time stamped and are sorted and processed in chronological order. Discrete-Event Systems are backed with formal mathematical descriptions [22] that facilitate formal verification and the construction of deterministic systems. Though these systems are good for real time applications the primary disadvantage is the computational cost of sorting the events globally to maintain the chronology.

2.1.2.4. Petri Nets

Petri-Nets [23] is a graphical representation introduced by Carl Adam Petri. With mathematical formalisms and abstraction, 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 = {p1, p2, p3, …. ,pm } is a finite set of places

T = {t1, t2, t3, …….,tm } is a finite set of transitions

F is a subset of (P x T) U (T x P) is a set of arcs giving flow relations

W: F -> {1, 2, 3, … } is the weight function

Mo: P -> {0, 1, 2, ….} 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.

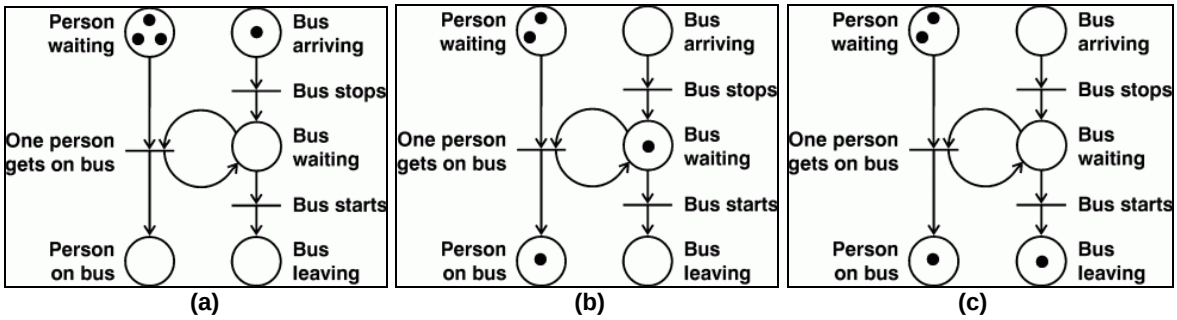


Figure 1. An Example Petri Net

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 feature of Petri Nets is its concurrent, asynchronous nature. There are a number of mathematical analyses that can be performed on Petri Nets.

The lack of hierarchy makes Petri Nets difficult to be used for large systems. Hierarchical Petri Nets (HPNs) [24] 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 [7]. This approach extends the Petri Net semantics with hierarchy making it suitable for complex systems.

2.1.2.5. 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.

Various flavors of data flow are seen in literature. 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 [27]. A static schedule is one that can be computed offline, has a finite sequence of execution of the nodes and required bounded buffers where the maximum size of the buffers is known before hand. 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 hence, it has a greater run-time cost. However, it can be used to represent a vast majority of systems and is more flexible than SDF.

Furthermore, many extensions have also been proposed to augment the data flow representation with hierarchy, strong data typing of tokens and parameterized nodes. The primary features of data flow are its inherent parallelism and data-dominated semantics.

2.1.3. HIGH-LEVEL MODELS

High-level models capture 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, 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 making high-level languages such as UML become executable artifacts. An executable artifact is one where the models capture sufficient information to facilitate the synthesis of low-level details. This section will highlight UML as a widely used system-level modeling language.

2.1.3.1. Unified Modeling Language (UML)

Unified Modeling Language (UML) [33] 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 the 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 on the triangle is the supertype while the classes connected to the bottom

are subtypes. Associations between classes are depicted with a line between the classes. On the association line the roles the classes play and their cardinality can also be specified. An association class can be specified to characterize an association. Composition represents a special kind of association that implies a class is composed of instances of another class and the instances cannot exist outside the composed class. The composition is depicted with a line having a diamond towards the composed class. The role and cardinality of the composing class is also specified on the composition line. Cardinality specifies the allowable lower and upper limit in the number of objects that are part of the association. For example, a 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.

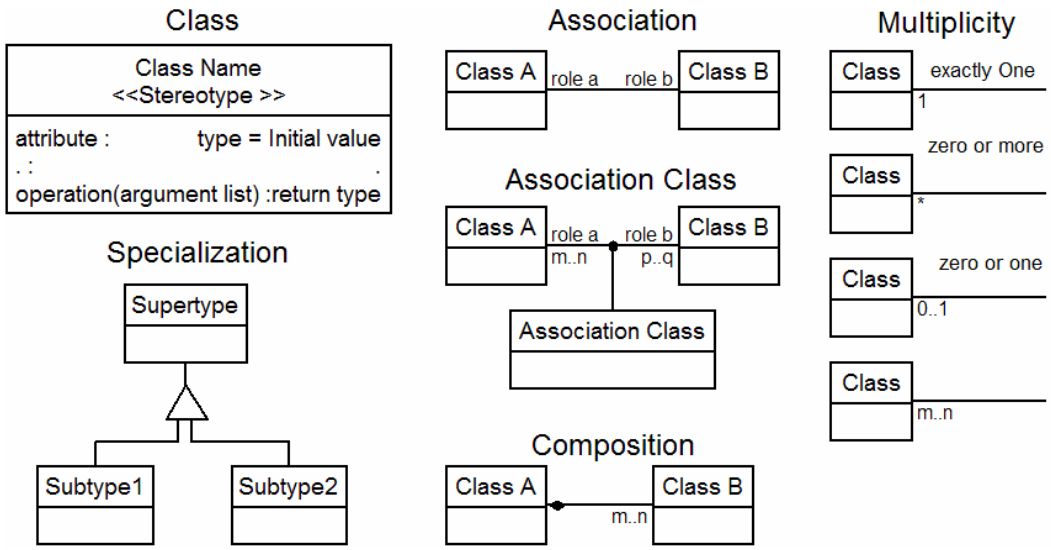


Figure 2 Basic notations of UML class diagrams [34]

2.1.4. LOW-LEVEL VS HIGH-LEVEL

In the previous section 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 devise. For example, 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 system 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 provided, 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 and time and effort.

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 doesn't span the entire computing space and/or one that makes a set of assumptions based on the domain.

2.2. 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 still 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.

2.2.1. 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 by providing a structured method for expressing knowledge about how a system, process, or organization works [28].

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 a 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. Unlike simulation languages (e.g., SIMAN, SLAM, GPSS, and WITNESS) that build predictive mathematical models, IDEF3 builds structured descriptions. 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 [28].

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 the 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." [28]

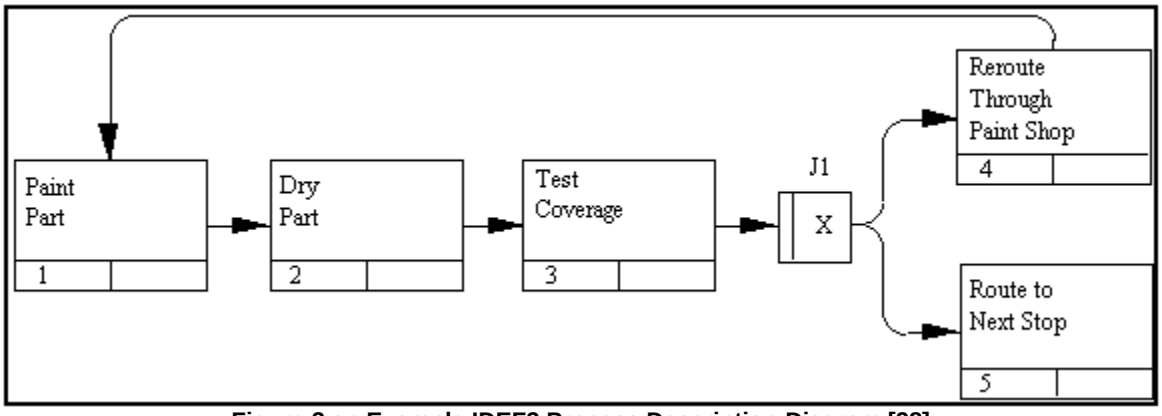


Figure 3 an Example IDEF3 Process Description Diagram [28]

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 five basic descriptive building blocks. [28]

2.2.2. PTOLEMY II – A POLYMORPHIC DESIGN ENVIRONMENT

Ptolemy is a project dedicated to the modeling, simulation and design of real-time, embedded applications started in 1990 at University Of California at Berkeley. The focus of Ptolemy is on component-based design. The philosophy of this project is centered on using different models of computation and developing an environment that allows the mixing of these models of computation to create a heterogeneous application [31].

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 [29].

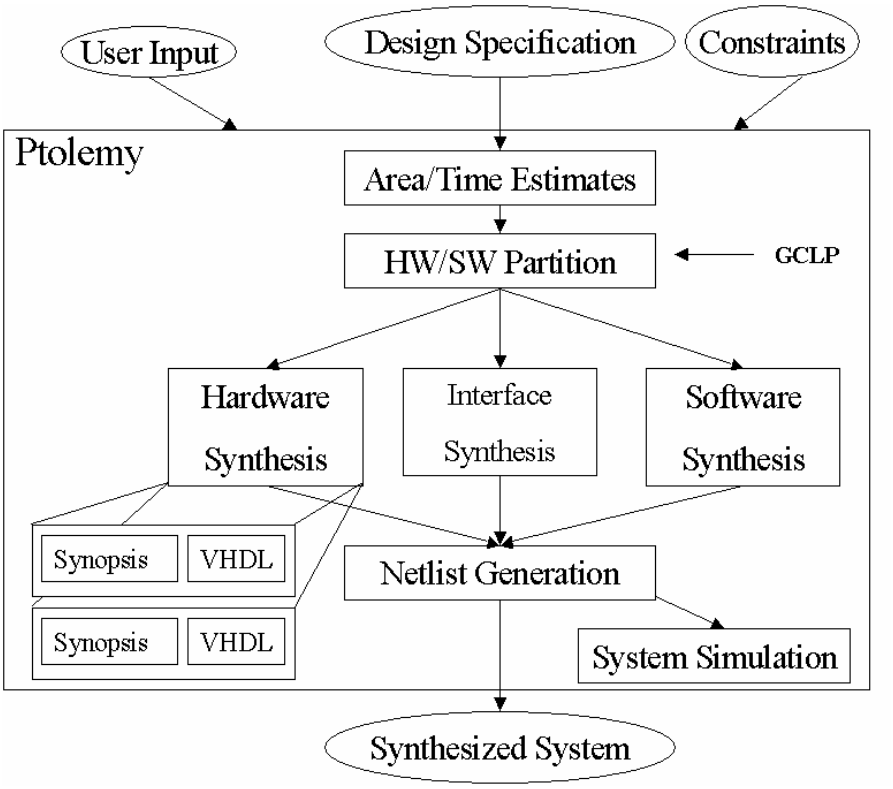


Figure 4 Design Methodology Management using Ptolemy [29]

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. After specifying the application using heterogeneous models, the next step is to partition the application. This is done using different partitioning algorithms like GCLP [30]. 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 [31].

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 [31].

2.2.3. 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. It is well know that GPLs have been more prevalent and successful compared to DSLs, even though

claims about DSLs' capabilities to increase productivity are widely accepted [35]. 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.

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 [36] used for simulation and control engineering. We believe that a mixed textual and graphical notation can be helpful in limited domains. For example, in the software development domain, the UML [33] specification has both textual (Object Constraint Language) and graphical (Use-Case Diagram, Class Diagram, etc.) notations. In hardware development domain, tool vendors [37] are now providing a graphical notation for the structural description of hardware while 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.

3. GENERATIVE AND MODEL BASED SOLUTIONS

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

Compiler compilers such as LEX [38] and YACC [39] 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).

3.1. 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.*
>
> *[40]*

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 [40].

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 is implemented [40].

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 come by means of parameters, programming using abstract types and algorithms written in a generic manner such that they can be reused. Meta programming can be viewed as a special case of generic programming where the programs are written with a lot of variability built in and the variability can be configured at various 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 people 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.

3.1.1. 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 [41][42].

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 to make 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* [41][42].

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 [41][42].

3.1.2. GENVOCA

GenVoca is a tool and methodology developed by Don Batory. The tool is based upon step-wise refinement of the domain specific specification. The next generation of the tool suite called AHEAD (Algebraic Hierarchical Equations for Application Design) [43] 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.

3.1.3. SUMMARY

Generative programming techniques are relevant and useful 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.

3.2. 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].

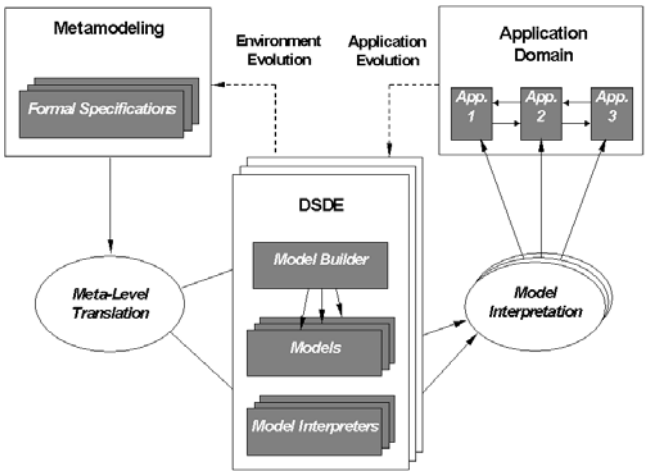


Figure 5 The MIC Development Cycle [1]

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 consideration of 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 [7]. 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 [13], Atom3 [14], DOME [15], Moses [16]. 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.

3.2.1. 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 doesn’t 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 and have then be triggered GUI on 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*.

3.2.2. 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 [44].

Moses has a textual metamodeling language called Graph Type Definition Language (GTDL) [46]. The language is used to specify formalisms (modeling language). GTDL allows the specification of the 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 [46].

Visualization information of the object is also declared, such as shape, border color, fill color and dimensions [45].

Moses support for static semantic constraints or the lack thereof is not clear from either the documentation or direct experimentation with the tool [45].

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 [45].

Moses supports animation of models by providing an extensible base animator class as part of the by the framework [45].

3.2.3. ATOM^3

Atom^3 is a multi-paradigm modeling tool developed at Modeling, Simulation and Design Lab (MSDL) in the School of Computer Science of McGill University. As other MIC implementations it also supports metamodeling.

The metamodeling language used by Atom^3 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 [47][48].

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 [47][48].

Dynamic semantics are represented using a graph grammar based transformation specification. The specification is converted to a python implementation [47][48].

3.2.4. KENT MODELING FRAMEWORK

The Kent Modeling Framework (KMF) is under development at the Computing Laboratory, University of Kent at Canterbury. KMF uses UML 1.3 and XMI 1.0 as the metamodeling language. UML class diagrams and constraints are fed to ToolGen that 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 [49].

3.2.5. 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 [50][51].

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 [50][51].

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 [50][51].

3.2.6. 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. 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 [33] 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 it makes GME behave like a specialized modeling environment supporting the target domain. Note that 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 and 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 [5].

3.2.7. COMPARISON OF FEATURES

Table 1: Comparison of Various MIC Tools

		DOME	Moses	Atom^3	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 complier	C++ compiler

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. Atom^3 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.

3.3. SUMARY 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 over come in order to have a greater impact on the software development community.

To enhance the development of model transformers, we need a way to precisely specify the operation of those transformers on categories of models, and to then generate the model transformer code from the specification. 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. It may be possible to use graph theoretic research and apply it to this problem.

4. GRAPH GRAMMARS AND TRANSFORMATIONS

Graph transformations and grammars have been a topic of research for well over 25 years. The 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 [52][53] and hyperedge replacement grammars [57][58]. 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 [60], single pushout [61] and programmed structure replacement systems [62]. The prominent graph transformation tools are AGG [66]and PROGRES [65].

This chapter is organized into sections where each section covers a particular class of graph grammars or transformation systems. Section 1 discusses node replacement grammars, followed by hyper-edge replacement grammars in Section 2. Section 3 deals with algebraic approaches while Section 4 discusses programmed graph replacement systems.

4.1. NODE REPLACEMENT GRAPH GRAMMARS

Node replacement grammars [52][53] 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 [52].

4.1.1. NODE LABEL CONTROLLED (NLC)

NLC is one of the first node replacement grammars that appeared in literature. NLC is defined as 5-tuple [52].

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

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 form 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 [52].

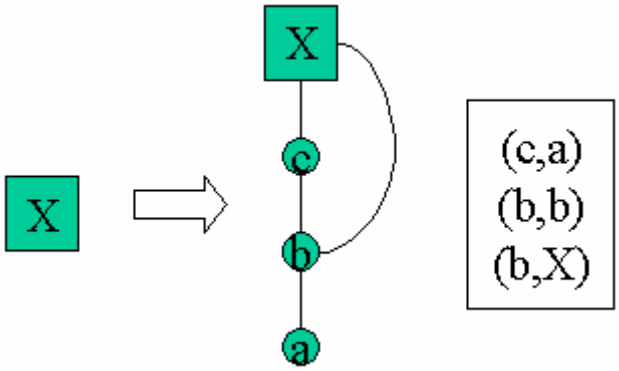
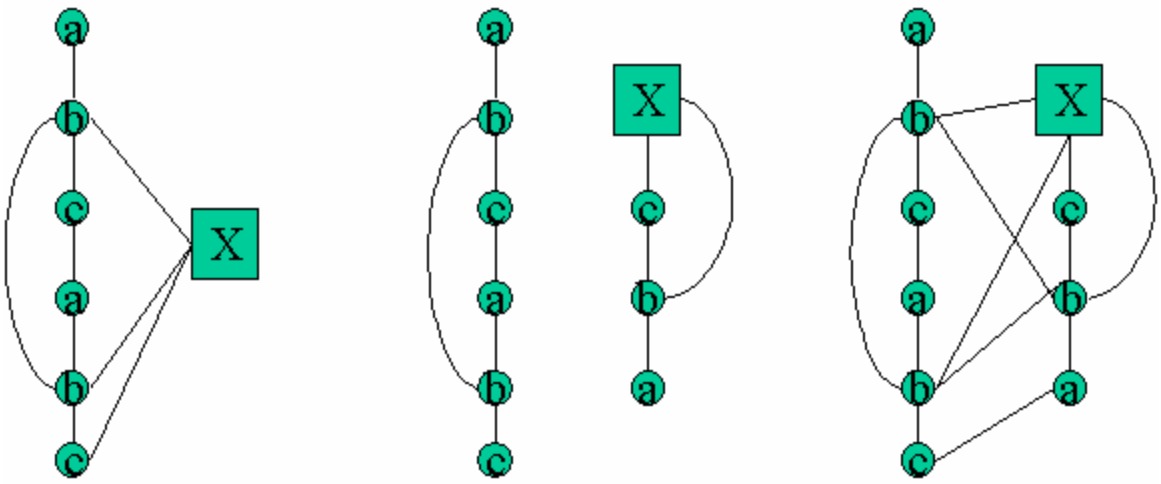


Figure 6 A NLC production

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) [52].



(a) (b) (c)

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 an 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 [52].

4.1.2. 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 → (D,C) where X → 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 the daughter graph.

NCE can be extended with direction on the edges. This gives rise to dNCE. By 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 [52].

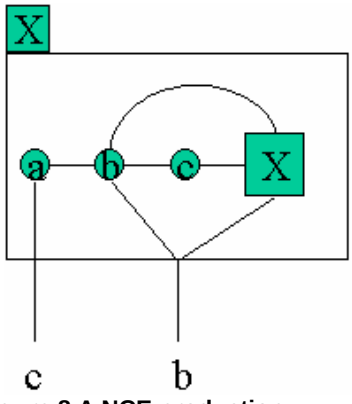


Figure 8 A NCE production

In an edNCE system the order of application of the production rules is unspecified. Thus different application sequences may lead to different graphs [52].

For this reason a property called confluence [54] 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 [52].

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
$\lambda : V \to \Sigma$ is the node labeling function.
The components of H are denoted as V_H, E_H and λ_H

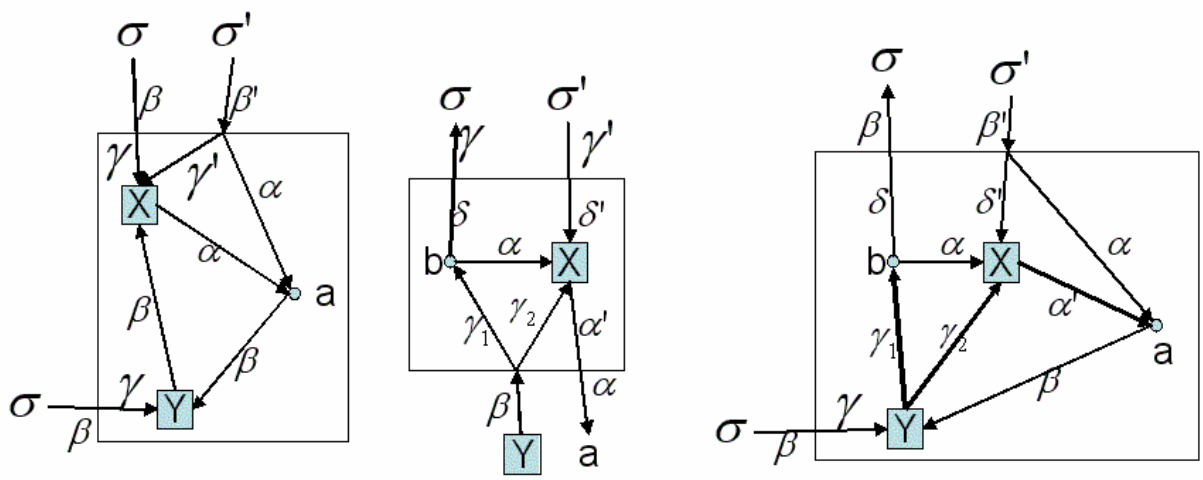
A production of edNCE grammar is of the form
$X \to (D, C)$ where
X is a non - terminal node label,
D is a graph and
C is a set of connection instructions. $C \subseteq \Sigma \times \Gamma \times \Gamma \times V_H \times \{in, out\}$
for readibility $C = (\sigma, \beta/\gamma, x, d)$ which means that 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 labaled σ with edge labeled γ and direction d

Let $\sum = \{X, Y, a, b, \sigma, \sigma'\}$ 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 → (H, C_H) and P2: X2 → (D, C_D) be productions in a grammar where, $C_D = \{(\sigma, \gamma / \delta, y, out), (\sigma', \gamma' / \delta', y, in), (Y, \beta / \gamma_1, y, in), (Y, \beta / \gamma_2, y, in), (a, \alpha / \alpha', y, out)\}$ and $C_H = \{(\sigma, \beta / \gamma, u, in), (\sigma, \beta / \gamma, u, out)\}$. 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.

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

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



(a) (H, C_H) (b) (D, C_D) (c) (H, C_H)[v/(D, C_D)]

Figure 9 Two graphs with embedding and the result of their substitution [52]

The above definition of embedding has been proven to be associative in [55], thus $K[w/H][v/D] = K[w/H[v/D]]$. Another important property of composition of productions is confluence which was proved in [56].

4.2. HYPEREDGE REPLACEMENT GRAPH GRAMMARS

Hyperedge replacement graph grammars [57][58] represents the next class of grammars that we study. 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.

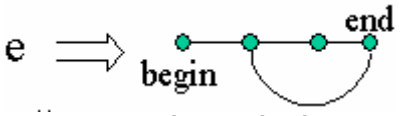
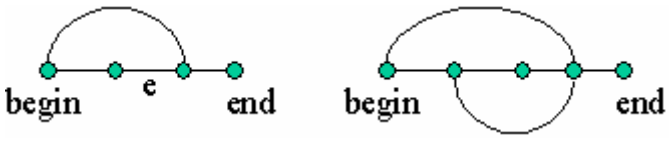


Figure 10 Hyperedge production

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).



(a) (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 doesn't affect the result. Finally, associativity 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 $e_1, e_2 ... e_n \in E_H$ and let H_i be the hypergraph. Then

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

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

1. **Context-freeness lemma;** it states that hyperedge replacement grammars are context free.
2. **Fixed-point theorem;** it states that hyperedge replacement grammars are the least fixed points of their generating productions.
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 [57].
4. **Parikh's theorem;** the theorem is originally for context free languages and had 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 [59].

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.

4.3. ALGEBRAIC APPROACH TO GRAPH TRANSFORMATION

The next approach to graph grammars is the algebraic approach [60][61]. 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) [60] and (b) Single PushOut (SPO) [61] 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.

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→V and destination d: E→V define a relation between vertices and edges. There are labeling functions lv: V→L_V and le: E→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, which deleted and which created new. The first step of applying a production is to match the LHS of the production in the host graph. A match m: L→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 LHR 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 [52].

4.3.1. DOUBLE PUSHOUT (DPO)

The basic approach in double pushout is: start with a subgraph to match in the input graph. Then an inverse gluing condition applied followed by a gluing. Put simply, two graphs are drawn. The production is represented as P => L ← K → 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.

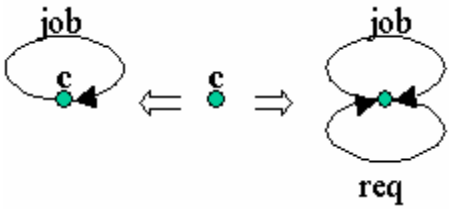


Figure 12 DPO production

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 $P1 \oplus_{p0} P2$ 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 the different processors $G_1 \oplus_{G_0} G_2$ and then the common part G0 is used to put them together.

4.3.2. 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 isn't. In SPO higher precedence is given to the deletion and thus in such cases the node will be deleted as shown in Figure 13.

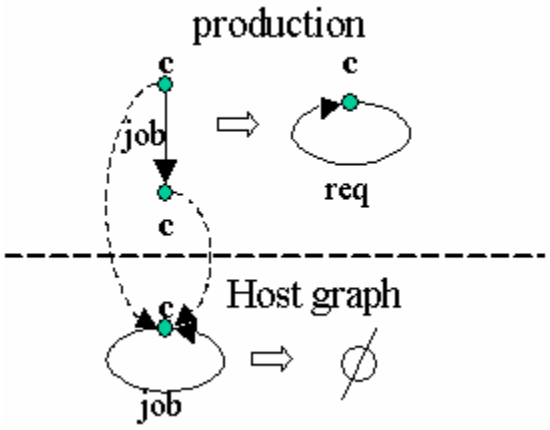


Figure 13 SPO production and example

4.4. PROGRAMMED GRAPH REWRITING SYSTEMS

The last section in this chapter is on programmed rewriting systems [62]. 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 [65]).

This section is broken down into three parts. The first part 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 section discusses PROGRES's approach to programmed graph rewriting.

4.4.1. LOGIC-BASED STRUCTURE REPLACEMENT SYSTEM

This section [62] critiques graph replacement system and the points out their deficiencies. The problems with other 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 states 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 logcal 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

$A_F := \{child, woman, wife, man, person, income, \text{int}, 0, ..., +\}$

$A_P := \{node, edge, attr, type, ...\}$

$\nu := \{He, She, Value, ...\}$

$\omega := \{HerChildren, HisChildren, ...\}$

$\chi := \{x1, x2, ...\}$

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),… , attr(Adam, income, 5000), attr(Eve, income, 10000), …., edge(Adam, wife, Eve), edge(Eve, child, Sally), . }

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

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

A Structure replacement rule is defined as a quadruple

$p := (AL, L, R, AR)$

$AL, AR \in F(\Sigma)$ where F(Σ) is the set of all negative conditions

$L, R \in \lambda(\Sigma)$ where $\lambda(\Sigma)$ is the set of all structures

For the example in Figure 14 the production will be

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

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.

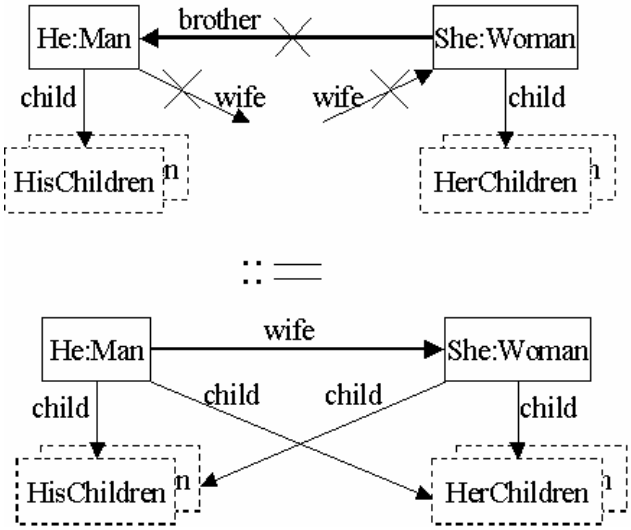


Figure 14 A production in the PROGRES system

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 edges 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.

4.5. 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 [63] and extended by Nelson in [64] 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.

4.6. SUMARY OF GRAPH GRAMMARS AND TRANSFORMATIONS

This chapter discussed the various graph grammars and transformations 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 that 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 languages. 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 use 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. Manipulation of different graphs that conform to different type systems is not allowed.

5. GRAPH TRANSFORMATION BASED TOOLS

The theory described above has given rise to many tools. Prominent amongst these tools are PROGRES, AGG, DiaGen and GenGEd. Out of these tools PROGRESS and AGG support general purpose graph transformation languages while DiaGen and GenGEd are visual language environments.

5.1. PROGRES

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

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 [62][65].

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 [62][65].

5.2. 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 doesn't have support for semantic constraints, such as 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 [66][67][68].

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. Alternately the user can disable the type graph and define node/edge types while creating the host graph [68].

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. Alternatively 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 [68].

5.3. 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	ATOM^3
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	??

5.4. GRAPH TRANSFORMATIONS CRITIQUE

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 rich mechanisms for controlling the application of transformation rules. PROGRES has a rich set of control mechanisms; however, they only perform transformations within the same domain. Domains specify the structural integrity constraints that the graphs must conform to. In PROGRES, these constraints are represented using schemas [65], while in AGG these are represented using type graphs [68].

In MIC, the 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:

$$\forall e1 \in E1 \Rightarrow \exists_1 e2 \in E2, \forall v2 \in V1 \Rightarrow \exists_1 v2 \in V2$$

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 the 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 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.

6. PROPOSAL

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 evolve and shortens the development cycle [1].

To leverage the benefits of MIC in Model Driven Architecture (MDA), the MDA scope needs to be expanded to Domain Specific MDA (DSMDA) where the focus is on developing the MDA process for specific domains. A particular DSMDA will 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 [5]. 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. Section 3.2 lists a set of basic feature the framework should support.

Tools such as GME [13], Atom^3 [14], DOME [15] and Moses [16] 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 (DSPSM).

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 it 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 it the bottleneck in the MIC process.

To speed up the development of DSMDAs a high-level language it required for the specification of model transformers. An execution framework can then be used to execute sentences of the language. Design of such a language is non-trivial as a model transformer can work with arbitrarily different domains and can perform fairly complex computations.

Design of such a language is non-trivial as a model transformer should work with arbitrary 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 we see that models in MIC are graphs, to be more precise they are typed multi-graphs. Thus, the model transformation problem can be converted into a graph transformation problem. We can then use the mathematical concepts of graph transformations [52] to formally specify the intended behavior of a model interpreter.

In Chapter 4. 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 [71][72][73][74]. Many tasks in software development can be formulated using this approach, including weaving of aspect-oriented programs [75] application of design patterns [73], and the transformation of platform-independent models into platform specific models [5].

These techniques have been developed mostly for the specification and recognition of graph languages, and performing transformations within the same "domain" (i.e. graph), while for model transformations the transformer should be able to work on two different kinds of graphs. Moreover, these transformation techniques use proprietary languages for the specification of the type systems and transformations. In summary, the following features are required in the transformation language:

1. Transformations are often required to convert graphs belonging to one domain/type system into graphs that belong to another domain/type system. The language should provide the user with a way to specify the different graph domains being used.

2. The transformation language should incorporate the graph domains in the transformation specification. It should use domain information to ensure type safety of graphs/models and guarantee that the transformation will preserve the syntactic correctness of the model/graphs.

3. The transformation language should be powerful enough to specify any kind of mapping. It should also have simple constructs that are easy to use.

4. There should be support for transformations that create independent models/graphs conforming to different domains. In the more general case there can be n input model/domain pairs and m output model/domain pairs.

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

6. All the previous points aim to increase productivity and achieve speed up in the time required for writing model interpreters. This is the primary goal and the most important aspect of this proposal.

6.1. RESEARCH HYPOTHESIS

"A Metamodel based transformation language using graph rewriting and transformations with support for multiple input and output graphs (corresponding to different domains) with an efficient implementation is suitable for the specification of model transformers. Such a language should help achieve a speed-up (the order of 2 to 10) in the time to develop model transformers and thus help develop a family of Domain Specific Model Driven Architecture tools."

6.2. RESEARCH METHODS

A new language is required that leverages the benefits of graph grammars and transformations and also provides the required language constructs for the development of model transformers. language development is based on gathering requirements of model transformations and then researching how these needs can be fulfilled by simple and formal constructs. Requirements should be gathered by looking at various target applications and by creating a list of challenge problems. Two challenge problems have been chosen:

1. Generate a non-hierarchical Finite State Machine (FSM) [8] from a Hierarchical Concurrent State Machine (HCSM) representation similar to Statecharts [20]. This problem introduces interesting challenges. To map concurrent state machines to a single machine there is a need for complex operations that include 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 [88]. 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 another interesting problem to try.

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 will focus on the execution framework for the language. Initially a working, non-optimized framework will be used to test the language constructs. Such a framework should be easily modifiable to try out different constructs and algorithms. Such a framework can then be used to test algorithms that are needed for the execution of sentences of the language.

6.3. COMPLETION CRITERIA

The language developed should successfully solve the challenge problems described above. Success will be measured using three metrics.

The first criterion is based on 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.

The second is the time taken to solve the problem in the new language compared to the traditional approach (hand written code). This will give an idea of the speed up achieved in development time. Experiments to quantitatively validate such a claim are difficult and so a qualitative measure will be used based on the feedback of people using the tool. A rough comparison will be made based on the results of similar work using traditional approaches.

The third metric is the speed of the generated model transformer compared to the hand-written model transformer. The language along with the execution framework should allow users with little programming knowledge to specify model transformers, achieve a speed up (greater than 2) in the time to develop these transformers and should not degrade the performance of the transformers by more than a constant factor.

6.4. PRELIMINARY WORK AND RESULTS

At present I have developed a language that satisfies the requirements of mentioned in the research proposal. The GReAT language is Turing complete and has simple programming constructs. A formal specification of the GReAT semantics has been described using the Zed notation [87]. The execution engine GReAT E has also been developed and can execute the transformations expressed in GReAT. The language has been successfully used to implement the both the challenge problems. Table 1 shows a comparison of specification in GReAT vs. hand code. The primitive rules are rules that contain graph transformations while compound rules are higher-level control flow constructs. The compassion gives a rough idea of how the transformations relate to hand code. However, better experiments are required to provide concrete results.

Table 3: Comparison of GReAT specification VS hand code

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

The execution engine needs to be efficient and some algorithms have been developed for pattern matching and efficient implementation of the language.

7. 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] 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.

[6] Bruno G., "Model Based Software Engineering", Chapman & Hall, 1995.

[7] "Model of Computation", Dictionary of Algorithms and Data Structure, National Institute of Standards and Technology, URL = http://www.nist.gov/dads/HTML/modelofcompu.html.

[8] "Finite State Machine", Dictionary of Algorithms and Data Structure, National Institute of Standards and Technology, URL = http://www.nist.gov/dads/HTML/finiteStateMachine.html.

[9] K. L. McMillan, "Symbolic Model Checking: an approach to the state explosion problem", CMU Tech Rpt. CMU-CS-92-131.

[10] "Turing Machine", The Stanford Encyclopedia of Philosophy (Summer 2003 Edition), (ed.), URL = http://plato.stanford.edu/archives/sum2003/entries/turing-machine/.

[11] "The Church-Turing Thesis", The Stanford Encyclopedia of Philosophy (E. Zalta, Edition), (ed), URL = http://plato.stanford.edu/entries/church-turing/.

[12] E. A. Lee, http://ptolemy.eecs.berkeley.edu/~eal/ee290n/glossary.html, EE290N: Advanced Topics in System Theory, Fall, 1996.

[13] A. Ledeczi, et al., "Composing Domain-Specific Design Environments", Computer, Nov. 2001, pp. 44-51.

[14] 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.

[15] "Dome Guide", Honeywell, Inc. Morris Township, N.J, 1999.

[16] Kim Mason, "Moses Formalism Creation – Tutorial", Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, CH-8092, Switzerland, February 9, 2000.

[17] 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.

[18] 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.

[19] 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.

[20] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231-274, June 1987.

[21] C. G. Cassandras, "Discrete Event Systems: Modeling and Performance Analysis", *Irwin Publications*, Boston, MA, 1993.

[22] 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.

[23] J. Peterson, "Petri Net Theory and the Modeling of Systems", *Prentice-Hall*, *Englewood Cliffs, NJ,* 1981.

[24] 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.

[25] T. De Marco, “Structured Analysis and System Specification”, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[26] C. P. Gane and T. Sarson, “Structured System Analysis: Tools and Techniques”, Prentice-Hall International, Englewood Cliffs, NJ, 1979.

[27] 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.

[28] R. J. Mayers, et all, “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.

[29] A. Kalavade, Edward A. Lee, “Design Methodology Management For System-Level Design”, Ptolemy Miniconference, March 10, 1995.

[30] 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.

[31] Edward A. Lee, “Overview of the Ptolemy Project”, *Technical Memorandum* UCB/ERL M01/11 March 6, 2001.

[32] P. P. Chen. "The Entity-Relationship Model". ACM Trans. on Database Systems (TODS), 1:9-36, 1976.

[33] J. Rumbaugh, I. Jacobson, and G. Booch, “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1998.

[34] M. Fowler, “UML Distilled Second Edition”, Addison Wesley Longman, Inc., 200.

[35] 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.

[36] “Simulink Reference”, The Mathworks, Inc., July 2002.

[37] ActiveHDL, http://www.aldec.com/ActiveHDL/, Aldec Inc., Henderson, NV 89074.

[38] M. E. Lesk, “LEX---a lexical analyzer generator”, CSTR 39, Bell Laboratories, Murray Hill, NJ, 1975.

[39] Johnson S.C., “Yacc: Yet Another Compiler-Compiler”, Bell Laboratories, Murray Hill, NJ, 1978.

[40] K. Czarnecki, U. Eisenecker, “Generative Programming: Methods, Techniques, and Applications”, Addison-Wesley, 1999.

[41] J. Neighbors, “Software Construction Using Components”, Ph.D. Thesis, ICS-TR-160, University of California at Irvine, 1980.

[42] J. Neighbors, “Draco 1.2 Users Manual”, University of California at Irvine, 1983.

[43] Don S. Batory, Jacob Neal Sarvela, Axel Rauschmayer, “Scaling Step-Wise Refinement”, International Conference on Software Engineering, pp 187-197, 2003.

[44] The Moses Project, Computer Engineering and Communications Laboratory, ETH Zurich URL = http://www.tik.ee.ethz.ch/~moses/

[45] R. Essar, J. Janneck and M. Naedele, “The Moses Tool Suite - A Tutorial”, Version 1.2, Computer, Engineering and Networks Laboratory, ETH Zurich, 2001.

[46] J. Janneck, “Graph-type definition language (GTDL)—specification”, Technical report, Computer, Engineering and Networks Laboratory, ETH Zurich, 2000.

[47] 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.

[48] 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.

[49] S. Kent, O. Patrascoiu, “Kent Modelling Framework Version – Tutorial”, Computing Laboratory, University of Kent, Canterbury, UK, Draft, December 2002.

[50] “ABC To Metacase Technology”, White Paper, MetaCase Consulting, Finland, August, 2000.

[51] “Domain-Specific Modelling: 10 Times Faster Than UML”, White Paper, MetaCase Consulting, Finland, January, 2001.

[52] Grzegorz Rozenberg, “Handbook of Graph Grammars and Computing by Graph Transformation”, World Scientific Publishing Co. Pte. Ltd., 1997.

[53] M. Nagl, “Formal Languages of Labeled Graphs”, Computing 16 (1976), 113-137.

[54] 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.

[55] G. Rozenberg, E. Welzl, “Graph Theoretic closure properties of the family of boundry NLC graph languages”, Acta Informatica 23, 289-309, 1986.

[56] R. Schuster, “Graphgrammatiken und Grapheinbettungen”, Algorithmen und Komplexitat, Technical Report MIP-8711, Universitat Passau, 1987.

[57] Annegret Habel, “Hyperedge Replacement: Grammars and Languages”, volume 643 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1992.

[58] Annegret Habel, “Hypergraph Grammars: Transformational and algorithmic aspects”, Journal of Information Processing and Cybernitics EIK, 28:241-277, 1992.

[59] R. J. Parikh, “On context-free languages”, Journal of ACM, 13:570-581, 1966.

[60] H. Erig, M. Pfender, and H. J. Schneider, “Graph Grammars: an algebraic approach”, In Proceegings IEEE Conf. on Automata and Switching Theory, pages 167-180, 1973.

[61] M. Lowe, “Algabraic approach to single-pushout graph transformation”, Theoritical Computer Science, 109:181-224, 1993.

[62] Blostein D., Schürr A., ”Computing with Graphs and Graph Rewriting”, Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany.

[63] E. W. Dijksrta, “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”, Communications of ACM, 18:453-457, 1975.

[64] G. Nelson, “A Generalization of Dijksrta’s Calculus”, ACM transactions on Programming Languages and Systems, Vol. 11, No. 4, pp-517-561, 1989.

[65] A. Schürr, “PROGRES for Beginners”, Technical Report, Lehrstuhl für Informatik III, RWTH Aachen, Germany.

[66] H. Gottler, “Attributed graph grammars for graphics”, H. Ehrig, M. Nagl, and G. Rosenberg, editors, Graph Grammars and their Application lo Computer Science, LNCS 153, pages 130-142, Springer-Verlag, 1982.

[67] H. Göttler, "Diagram Editors = Graphs + Attributes + Graph Grammars," International Journal of Man-Machine Studies, Vol 37, No 4, Oct. 1992, pp. 481-502.

[68] C. Ermel, T. Schultzke, “The Agg Environment: A Short Manual”, TU Berlin.

[69] J. Loyall and S. Kaplan, "Visual Concurrent Programming with Delta-Grammars," Journal of Visual Languages and Computing, Vol 3, 1992, pp. 107-133.

[70] 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.

[71] 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.

[72] 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.

[73] A. Radermacher, ``Support for Design Patterns through Graph Transformation Tools", Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, The Netherlands, Sep. 1999.

[74] A. Bredenfeld, 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.

[75] H. Fahmy, B. Blostein, “A Graph Grammar for Recognition of Music Notation”, Machine Vision and Applications, Vol. 6, No. 2 (1993), 83-99.

[76] 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).

[77] G.Schmidt, R. Berghammer (eds.), “Proc. Int. Workshop on Graph-Theoritic Concepts in Computer Science”, (WG ’91), LNCS 570, Springer Verlag (1991).

[78] H.Ehrig, M. Pfender, H. J. Schneider, “Graph-grammars: an algebraic approach”, Proceedings IEEE Conference on Automata and Switching Theory, pages 167-180 (1973).

[79] Object Management Group, Object Constraint Language Specification, OMG Document formal/01-9-77. September 2001.

[80] A. Bakay, “The UDM Framework,” http://www.isis.vanderbilt.edu/Projects/mobies/.

[81] J. McCarthy “Recursive functions of symbolic expressions and their computation by machine – I”, Communications of the ACM, 3(1), 184-195, 1960.

[82] Uwe Assmann, “Aspect Weaving by Graph Rewriting”, Generative Component-based Software Engineering (GCSE), p. 24-36, Oct 1999.

[83] G. Karsai, S. Padalkar, H. Franke, J. Sztipanovits, ”A Practical Method For Creating Plant Diagnositics Applications”, Integrated Computer-Aided Engineering, 3, 4, pp. 291-304, 1996.

[84] E. Long, A. Misra, J. Sztipanovits, “Increasing Productivity at Saturn”, IEEE Computer Magazine, August 1998.

[85] AGG, http://tfs.cs.tu-berlin.de/agg/.

[86] 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.

[87] 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.

[88] T. A. Heinzinger, “The Theory of Hybrid Automata”, In Proc. Of IEEE Symposium on Logic in Computer Science, IEEE press, pp 278-292, 1996.