Generation of Synthetic Embedded Application Models Based on Meta-models

Adam Derda

Supervised by Paul Pop

Kongens Lyngby 2009
IMM-MSC-2009-35

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

Embedded systems are present everywhere: from alarm clocks to PDAs, from mobile phones to cars. Almost all devices that we use are controlled by embedded systems. Embedded systems work in different environments; they have to fulfill many different functional and non-functional requirements, such as cost, energy consumption, reliability and flexibility.

Hence, it is important to have tool support for successfully designing embedded systems. Modern embedded systems design relies on models for the application behavior and hardware platform. Starting from these models, the embedded systems system-level design tasks are responsible for finding a model of the implementation that can later be synthesized to hardware and software.

The quality of a design technique has to be evaluated using several case studies. Such case studies are often difficult to obtain, hence researchers use randomly generated synthetic application models. There are many types of models of computation and communication used in embedded systems, such as Data-Flow and Sequencing Graphs, Petri nets, Kahn Process Networks, and a separate synthetic-model software generation tool has to be written for each model.

The objective of this thesis is to create a software tool for the generation of synthetic application models for embedded systems. The tool is generic, i.e., it is able to take as input any meta-model describing the embedded system models that have to be generated. We have proposed a set of entities and attributes that are used to specify a broad range of embedded systems meta-models. Using these entities, the user can graphically describe the meta-model within the Generic Modeling Environment (GME), an open-source meta-modeling tool. The model generator produces (1) synthetic application models conforming to the meta-

model specification and (2) a meta-model for GME, such the generated models can be loaded into GME and further manipulated, if necessary.

The proposed tool eliminates limitations of the currently used generators — single model type support, limited configuration and difficult usage. The implemented solution has been verified using several case studies of the most widely used embedded system models.

Contents

Chapter 1

Introduction

Embedded systems are present everywhere: from alarm clocks to PDAs, from mobile phones to cars. Over 90% of microprocessors are used in embedded systems, the number of embedded systems in use has become larger than the number of humans on the planet, and is projected to increase to 40 billion worldwide by 2020. Almost all devices that we use are controlled by embedded systems. Embedded systems are the key to the competitiveness and innovation in many major European industries, including Danish industry [20].

An embedded system is a special-purpose computer system, part of a larger system which it controls. The main characteristics of embedded systems are that: they are single-functioned — dedicated to perform a single function; they have complex functionality — often have to run sophisticated algorithms or multiple algorithms, e.g., cell phone functionality; they are tightly-constrained — have to be low cost, low power, small, fast, etc.; they are reactive and real-time —continually reacting to changes in the system's environment and must compute certain results in real-time without delay; they are often safety-critical — must not endanger human life and the environment.

Hence, the design of an embedded system is a very challenging and complex task. Therefore, it is important to have tool support and use the right design methodology for successfully designing embedded systems. The aim of a design methodology is to coordinate the design tasks such that the time-to-market is

minimized, and the design constraints are satisfied. Traditionally, the hardware and software parts are developed independently, often by different teams located far away from each other. Software code is written, the hardware is synthesized and they are supposed to integrate correctly from the first attempt.

Such an approach does not work for today's systems. Modern embedded systems design relies on models for the application behavior and hardware platform. Starting from these models, the embedded systems system-level design tasks are responsible for finding a model of the implementation that can later be synthesized to hardware and software.

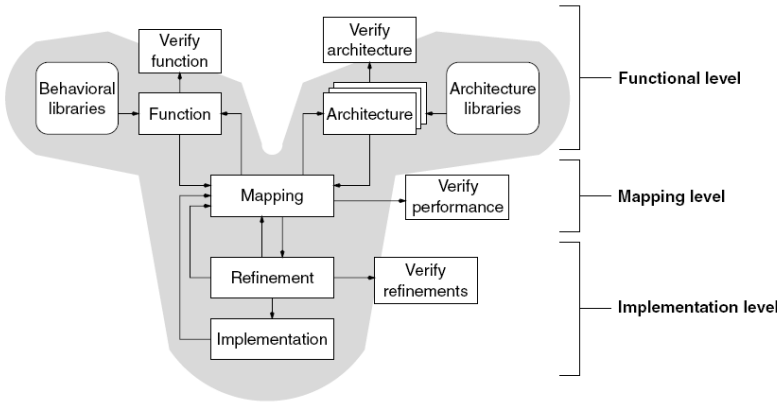


Figure 1.1: Embedded system design methodology. *Source: [35]*

A possible system-level design process is presented in figure 1.1. At the functional level, behavioral specification is designed and verified, as well as different hardware architectures, which may be used as behavior implementation, are created. On the mapping level different functionalities are assigned to different hardware components. By mapping the same behavioral specification to different kinds of hardware architecture designers can choose an optimal solution (with the best performance, the smallest space used, etc). On the implementation level the lower levels of abstraction are generated in a semi-automatic manner.

The design tasks are implemented as software tools, and they use different kinds of embedded system models, which provide a formal way to describe functionality of embedded systems. The quality of design tools for the successful design of embedded systems. Many companies are research groups, including the Embedded Systems Engineering section at DTU Informatics, are currently developing such state-of-the-art design tools.

After developing a tool, it must be thoroughly evaluated; its quality is typically determined by running them on case studies from the industry. Such case studies are often difficult to obtain, hence researchers use randomly generated synthetic application models. There are many types of models of computation and communication used in embedded systems, such as Data-Flow and Sequencing Graphs, Petri nets, Kahn Process Networks, and a separate synthetic-model software generation tool has to be written for each model.

The objective of this thesis is to create a software tool for the generation of synthetic application and platform models for embedded systems. The tool is generic, i.e., it is able to take as input any meta-model describing the embedded system models that have to be generated. The figure 1.2 presents the diagram of the proposed solution. Firstly, a set of entities and attributes (a meta-model) describing an embedded system meta-model is defined. Using these entities, the user can graphically describe the meta-model within the Generic Modeling Environment (GME), an open-source meta-modeling tool. The meta-model is then used as an input of the developed GME plug-in. It interprets the meta-model and transforms it to the designed data structure. The data structure is then used by generator to produce synthetic embedded system models in three different formats, as well as to generate GME meta-model. Having such a meta-model, user can load and modify synthetic models in GME environment or even create models manually.

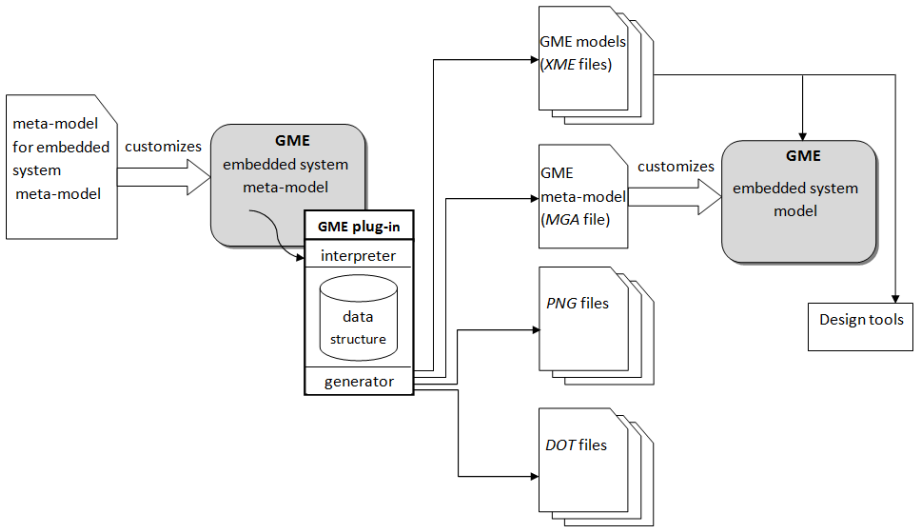


Figure 1.2: Process of model generation using GME and the Model Generator plug-in.

This thesis is organized in 7 chapters. Chapter 2 is a comparison of the most widely used models of embedded systems. The purpose for the chapter is to define specific requirements to the general meta-model of embedded system model. In chapter 3 there is a short overview of currently available solutions, which shows a need to develop a new, more general and model independent solution. Chapter 4 is a detailed description of a meta-model, which is used as an input of the implemented model generator. Chapter 5 contains an overview of the tool implementation. This chapter contains a description of the created meta-model interpreter, the model generator and details of the algorithm generating models. In chapter 6 an evaluation of the developed solution is performed. It is based on several case studies: generating directed acyclic graphs, Petri nets and sequencing graphs for biochips. Chapter 7 contains a summary of the entire thesis.

Chapter 2

Embedded Systems Modeling

A model of computation is an abstract representation of the behavior of an embedded system. It defines two device aspects [17]:

- components – description of each component functionality (e.g. procedures or finite state machines),
- communication protocols – description of interaction, between system components (e.g. randez-vous or asynchronous message passing.

Optionally, model of computation can specify information sharing between components (like global variables). Since embedded systems are used in many different domains, there are a lot of different models of computation. Depending on the abstraction level and application area, designers can choose models that describe system in the most detailed and appropriate manner.

This chapter is an overview of the models that are most widely used to describe system level of abstraction: task graphs (section 2.1), Petri nets (section 2.2), Kahn process network (section 2.3). The last section contains a presentation of a sequencing graph for biochips, which in not as popular as the previously listed ones, but it has a much more complicated structure. The main purpose of the comparison is to find common properties in graphical representation of

each embedded application model, which allow defining detailed requirements to the general meta-model of embedded system model. The summary of the performed analysis is a content of section 2.5.

2.1 Task Graphs

One of the most important issues related to the multi-processor systems is the problem of efficient task distribution, which shall minimize the execution time of the program. Solving this problem is crucial for the system to achieve high performance. However, it is strong NP-complete even in the simplest cases: assuming that time of every task is the same and system contains arbitrary number of processors, as well as in the situation, when there are only two types of task with different execution times running on two processors [14]. There are many different methods proposed in literature to solve task distribution problem in a reasonable period of time. Some of them are based on clustering similar tasks in order to make the problem as simple as possible for each cluster [34][24]. Other scientists categorize parallel system into different classes, depending on their structure and task characteristic and prepare optimal solution just for a chosen class [9][6]. Yet another approach is to find a dynamic critical path and planning execution according to the current and predicted nodes usage [14].

Correctness and efficiency of task distribution algorithm is verified using model of the application. This model is mostly represented as task graph, which is an example of directed acyclic graph (DAG). Vertices of the graph represent tasks. In all cases, there is an integer or a floating point number assigned to each node. This number represents time of execution of each task [18]. This time is usually not a constant, but random and has different kinds of distributions. The most commonly used are uniform, normal and exponential distributions [29].

Each edge in a task graph represents precedence between tasks. Figure 2.1 shows simple directed acyclic graph with 10 nodes (10 tasks). It is easy to see task dependencies on the presented graph. In this situation task 6 must be completed before tasks 7 and 8 are executed. Tasks 7 and 8 be executed concurrently. Task 9 may be started only if task 2, 4 or 5 is completed. Task 6 may start immediately after the execution of task 4, or after completion of task 5 inbetween.

Similarly to nodes, arcs may also have assigned weights. Those numbers represent communication cost between processes. In many models it is skipped and it is implicitly included in the execution time of a task [18].

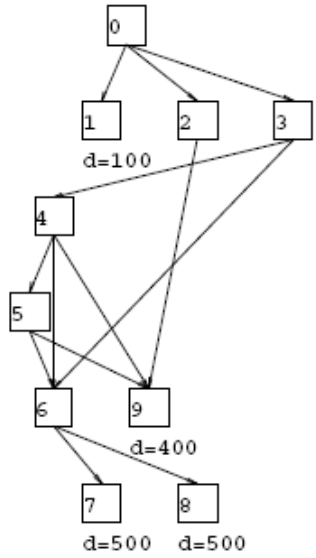


Figure 2.1: Example of directed acyclic graph with 10 nodes. *Source: [8]*

The arcs of a DAG cannot duplicate, which means that if there is one edge connecting vertex 0 and 1, there cannot be another edge connecting the same pare of vertices. Maximum number of edges outgoing from a node and incoming to a node is system dependent. Another parameter describing model is a number of tasks, i.e. the number of vertices.

2.2 Petri Nets

Another way of modeling complex systems is using Petri nets. Inventor of this mathematical and graphical tool is Carl Adam Petri. As a mathematical tool Petri nets are a set of linear equations that allows conducting formal proofs of model correctness, relation precedence, freedom from deadlock, etc. The detailed description of Petri net as a mathematical tool is beyond the scope of this thesis [19][36].

As a graphical tool Petri nets illustrates in a simple way behavior and dataflow of a system. They are widely used during designing of communication protocols, as well as models of different kinds of embedded systems (machine shops,

automated assembly lines, automotive industry etc.). Petri nets are used as an alternative to the ladder logic diagrams during designing PLC logic. They are also used by software engineers to model and analyze software behavior [36].

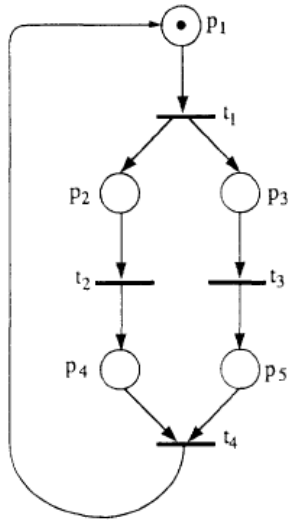


Figure 2.2: Example of a simple Petri net. *Source: [36]*

Petri net is a kind of a directed graph, where cycles are allowed. It contains two types of vertices: places (represented as circles) and transitions (represented as bars). Edges are directed and can lead from a place to a transition or from transition to a place. A place that has an edge outgoing towards a transition is an input place. A place which has an edge incoming from a transition is an output stage. Transitions represent events in a system, input places are their pre-conditions, whereas output places are post-conditions. Petri net elements may also be interpreted in the context of system resources. In such a case input state represents resource availability, transition represents its utilization and output place is resource releasing. Figure 2.2 shows an example of Petri net with 4 transitions and 5 places. Place p_3 is input place for transition t_3, and it is one of the two output places of transition t_1 [36].

There may be more than one edge between the same pair of states in Petri nets. Using multiple edges may be expressed as a number of parallel edges, as well as a label to a single edge. Label is then a nonnegative integer number indicating multiplicity of edges. In order to specify dynamic behavior of a system, tokens are used. Token is a nonnegative integer number assigned to a place (a place is marked with tokens). Graphically, each token is represented as a dot inside a

place. A transition is "enable" if each of its input places is marked with at least k tokens, where k is multiplicity of edges. When a transition is "enable", it may be executed – a required number of tokens is cleared from all the input places and all the output places are marked according to the multiplicity of outgoing edges [19].

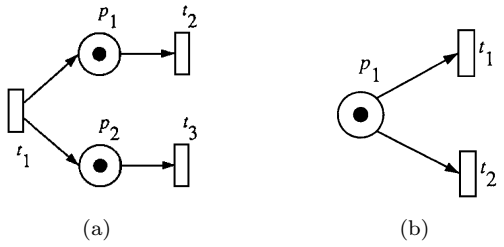


Figure 2.3: Concurrent vs. alternative transitions in Petri net. *Source: [19]*

Using Petri nets it is very easy to show concurrency and alternative paths of execution. In order to present that two events may happen concurrently, one transition leads to two different places. Both of them are input places for two different, concurrent transitions (as presented in figure 6.7(a)). To show choice between two paths, one place leads to two different transitions. In figure 6.7(b) there are two alternative transitions t_1 and t_2. Only one of them may be executed, when place p_1 is marked with one token.

2.3 Kahn Process Networks

Kahn process networks were proposed by Gilles Kahn in 1974 [12]. It is a model of computation for multiprocessor systems, mainly used for developing signal-processing applications. There are two elements in this model: processes and channels. Each process performs sequential computation. Processes communicate using channels, which are FIFO queues with unbounded capacity. Each process can read a channel, or write to a channel. Since channels have unbounded capacity, write operations are non-blocking. Read operation stalls the process until all the required input data is available in a channel. Kahn process network is a deterministic model, which means that the result of computation is always the same, for the same set of inputs, independently from the schedule used to calculate the result [13].

Kahn process network can be presented as a directed graph, where cycles are allowed. An example of a graphic representation of Kahn process network is

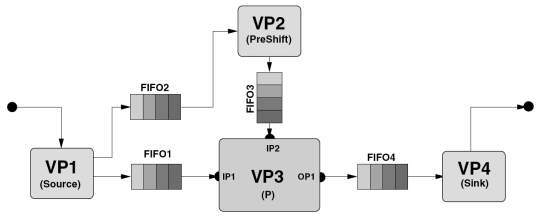


Figure 2.4: Example of a simple Kahn process network. *Source: [26]*

presented in figure 2.4. Nodes of the graph represent processes. Channels and directions of data exchange are presented as directed edge of a graph. Number of input and output channels for each process is system dependent.

2.4 Sequencing Graphs

There are two types of sequencing graphs: data flow graphs and control/data flow graphs. The first type ilustrates dependencies between data in a process, whereas control/data flow graphs represent control dependencies. Sequencing graphs are widely used in software engineering, to ilustrate process of software execution. They are also used by hardware designers to demonstrate control and data flow in circuits [3][5].

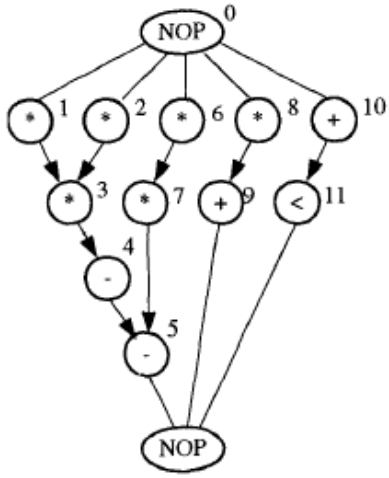


Figure 2.5: Example of a sequencing graph. *Source: [3]*

An example of a simple sequencing graph is presented in figure 2.5. It is represented by a directed graph. Each node represents an operation. Direction of an edge specifies the direction of a data/control flow. Edges may form cycles, to present loops in a model. In order to show dynamic behavior of a model, tokens are used. A process can be executed only if tokens are available in each of incoming edges. After execution, the process produces tokens for all its successors.

The structures of a sequencing graphs may be more complex. There may be many rules describing predecessor types, required for a specified type of node, types of starting and final types of nodes, etc. A good example of such a case is a sequencing graph for biochips, capturing the operations of a biochemical application. There are four different types of nodes: input, dilution, mixing and detection. Only input nodes may act as starting nodes of a graph. Only detection type nodes can act as final nodes. Each mixing node require two predecessors: 2 inputs, 2 mixings, 1 mixing and 1 input or 1 dilution and 1 input. Each dilution node may have one dilution type predecessor or two predecessors: both mixing, both input, one input and one mixing or one input and one dilution. Only detection type nodes can act as successor to the mixing node [33][32].

2.5 Summary

All the presented models are different types of directed graphs. A formal definition of a graph is (quote from [28]):

> A graph $G = (V, E)$ consists of two sets: a finite set V of elements called vertices and finite set E of elements called edges. Each edge is defined with a pair of vertices. If the edges of a graph G are identified with ordered pairs of vertices, then G is called directed or an oriented graph. Otherwise G is called an undirected or a nonoriented graph.

The main differences between the presented models are as follows:

- multiple types of vertices – each model uses different types of nodes, described by different types of parameters (number of tokens, execution time, etc.); some of the models contain a few different types of nodes (e.g. input, dilution, mixing and detection in sequencing graphs for biochips),
- multiple types of edges – a model may use more than one type of edge; e.g. in Petri nets there are two types of edges: one connecting a place

with a transition and one connecting a transition with a place (there is no connection between the same types of vertices); similarly to vertices, edges may be associated with different types of parameters (e.g. communication cost, multiplicity, etc.)

- cycles – some of the presented models are represented by acyclic graphs, whereas for some of them, cycles are allowed,
- multiplicity of edges – some models (like Petri nets) allow creating more than one edge between the same two nodes,
- minimum/maximum number of outgoing and incoming edges – this parameter is different for each system; it depends on the hardware/software architecture,
- type of start and final vertices – not all the types of vertices in a model may be used as start of final types; e.g. a Petri net may start only from place,
- multiple types of predecessors/successors – some models (like presented sequencing graphs for biochips) specify more than one type of successors or predecessors of the specified vertex type.

Chapter 4 contains the description of a set of entities and attributes, with associated constraints, which can be used to specify a broad range of meta-models describing embedded system models, such as the ones presented in this chapter.

Chapter 3

Related work

This chapter is an overview of the currently available model generators. Section 3.1 is a short presentation of Task Graph for Free DAGs generator. Section 3.2 describes SDF For Free tool, which is a generator of data flow graphs. Overview of a tool that extracts model from a C code files is presented in section 3.3. Section 3.4 contains a short summary of the chapter with emphasis being put on the answer to the question: what is the point of developing a new model generator?

3.1 Task Graphs for Free

Task Graph for Free (TGFF) is a generator of random directed acyclic graphs. It is able to generate graphs with completely random structure, as well as series-parallel graphs. The tool has no graphic interface. All the parameters are passed as the program's arguments. The basic settings allow user to specify [30]:

- number of graphs to be generated
- minimum number of vertices per graph
- number of start vertices

- mean execution time of each task
- additional parameters associated with each vertex or each edge
- maximum number of edges incoming to a vertex and outgoing from a vertex

The simple TGFF input script is presented below.

```
#num_task_graphs    2      3
#avg_tasks_per_pe   8      10
#avg_task_time      1000   1000
#mul_task_time      250    250
#task_slack         200    200
#num_pe_types       3      3
#num_pe_soln        4      4
#num_pe_com_types   1      1
#num_com_soln       1      1
#arc_fill_factor    0.2    0.1
#max_in/out_deg     3,3    5,5
```

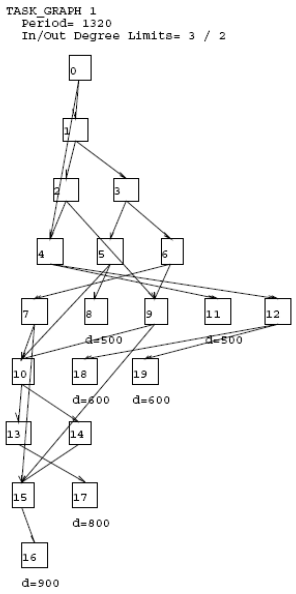


Figure 3.1: Example of TGFF output.

Task Graph for Free generates 3 types of output files: native *tgff* format file, PostScript file with plot of a graph and *vcg* file (input for Visualization of

Compiler Graphs tool). Example of a graph that was generated to a PostScript file is presented in figure 3.1.

3.2 SDF For Free

SDF For Free is a tool generating synchronous data flow graphs (SDFGs), with random structure. The tool is used to generate synthetic models of multimedia applications, which then can be mapped to a multiprocessor system. Similarly to TGFF, the tool has no GUI. A user has to define graph properties in a XML file instead. SDF For Free allows defining the following properties [27]:

- *actor* properties – properties of nodes, like name, minimum number of input/output ports, etc.
- *channel* properties – properties of edges, like source and destination type, initial token number, etc.
- maximum execution time
- maximum token number
- bandwidth and throughput attributes
- time constraints

Example of an input XML file is presented below.

```
<sdf3 type='sdf' version='1.0'
    xsi:noNamespaceSchemaLocation='http://www.es.ele.tue.
       nl/sdf3/xsd/sdf3-sdf.xsd'>
    <settings type='generate'>
        <graph>
            <actors nr='10'/>
            <degree avg='2' var='1' min='1' max='5'/>
            <rate avg='2' var='1' min='1' max='5'
               repetitionVectorSum='10'/>
            <initialTokens prop='0'/>
            <structure stronglyConnected='false' acyclic=
               'true' multigraph='true'/>
        </graph>
        <graphProperties>
            <procs nrTypes='3' mapChance='0.25'/>
```

```
            <execTime avg='10' var='0' min='10' max='10'/
                >
            <stateSize avg='1' var='1' min='1' max='1'/>
            <tokenSize avg='1' var='1' min='1' max='1'/>
            <bufferSize/>
            <bandwidthRequirement avg='2' var='0' min='1'
                 max='4'/>
            <latencyRequirement avg='2' var='0' min='1'
                max='4'/>
            <throughputConstraint autoConcurrencyDegree='
                1' scaleFactor='0.1'/>
            <integerMCM/>
        </graphProperties>
    </settings>
</sdf3>
```

SDF For Free generates graphs in a *DOT* format file. It can be transformed into a graphical representation using *Graphviz* tool. An example graph generated by the tool is presented in picture 3.2.

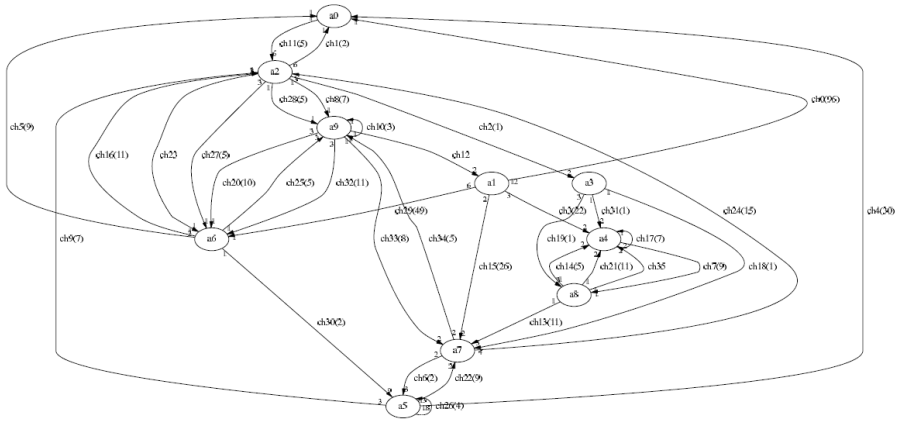


Figure 3.2: Example of SDF For Free output transformed to a graphical representation.

3.3 Model Extraction

Another approach to model generation is to extract it from an application. This technique is presented in [31].

Model generated by the tool is a directed acyclic graph. As an input it takes C code of a program. The code is transformed by a preprocessor. After that, abstract syntax tree of a program is generated. Next step is keywords extraction. A keyword is a piece of code, which allows identification of data dependencies. After analysis of keywords, a dependence graph is created. To get the DAG model from the dependence graph, only execution and communication times must be added.

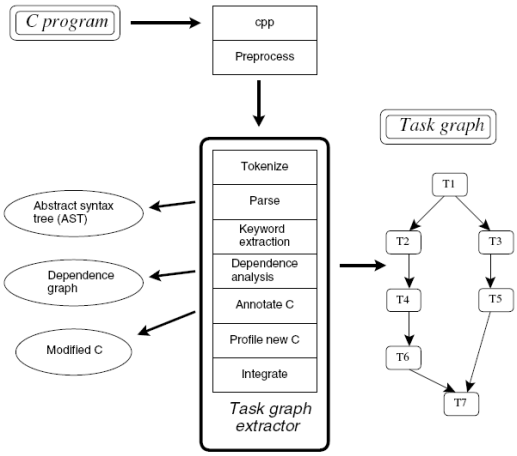


Figure 3.3: Process of a task graph extraction from C code files. *Source: [31]*

To extract execution and communication times, two modified C files are created. The first one contains annotations, which specify where each task begins and ends. The second file is used to profile the application. After the execution of the modified file, all the timing information is recorded. The timing data and the dependence graph, obtained in the previous steps, are used to generate final model of the application. A schema of the above described process is presented in figure 3.3.

3.4 Reason for Developing a New Solution

There are very few tools available, which are able to generate application models. Most of the researchers do not use publicly available model generators. They usually develop their own tools to generate a set of models dedicated to the specified system, which makes these tools useless for other researchers. Furthermore, model generators have to be developed from scratch, when model of computation is changed (e.g., from DAG to Petri net).

To make the benchmarking of tools easier, a new, generic, model-independent tool should be implemented. User should be able to freely change all the model parameters, types of vertices and connections, constraints on the number of vertex input and outputs, variables associated with nodes and edges etc. The details of the design and implementation of such a tool are described in the next chapters of this thesis.

Chapter 4

A Generic Meta-Model for Embedded System Models

The aim of the thesis is to create a generic model generator. A user should be able to freely define different properties of a model: different types of vertices and edges, parameters associated with them, restrictions on the number of incoming and outgoing edges, etc. This specification can be expressed in many different ways, e.g., using scripts with a number of different parameters or using XML files with model description. However, using a textual model description would make the tool less user friendly. It would require the user to be familiar with a large number of parameters or a complicated XML schema. A much better solution is to use a meta-modeling language and one of the meta-modeling tools which have a user friendly GUI.

This chapter contains a detailed description of the meta-model that is used as an input of the Model Generator (described in details in chapter 5). Section 4.1 of this chapter introduces basic meta-modeling terminology. Section 4.2 is an overview of available meta-modeling tools and our decision on which one to sue for this thesis purposes. Section 4.4 presents details of meta-model for embedded system models.

4.1 Basic Meta-Modeling Terminology

A **meta-model** is a model that specifies how a model is built. It defines all the entities, relations and attributes that model can be composed of.

A **generic modeling environment** supports different types of model domains. This make the tools universal, however, the general approach causes lack of domain specific details, which usually are crucial for model designers.

A **domain-specific modeling environment** may be used as an alternative. It is dedicated only to one, specific domain, so a designer can describe all the model details. However, domain-specific approach is a very expensive and ineffective solution – it requires developing many separated tools, just to model a small part of the entire modeled system. In order to combine advantages and reduce disadvantages of the both approaches, **meta-programmable tools** have been developed. They constitute generic frameworks, which have to be customized by the user, in order to make them domain-specific.

The process of meta-programmable tools customization is performed using a **meta-programming language**. After applying a configuration described in a meta-programming language, the modeling tool becomes domain-specific [16].

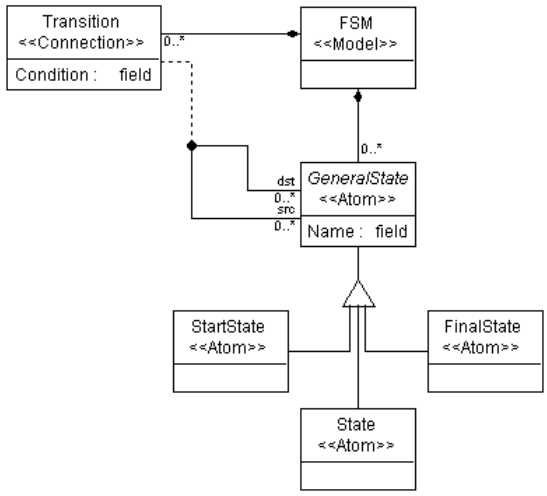


Figure 4.1: A finite-state machine meta-model

An example of a finite-state machine meta-model is presented in figure 4.1. It is described in UML, using the Generic Modeling Environment (GME) tool,

which is a meta-programmable tool. The meta-model is used for transforming the generic environment into domain-specific modeling environment. According to the defined specification a model may be composed of *StartState*, *State* and *FinalState* entities. They may be connected with each other using *Transitions*. An example of a FSM model, created in GME environment is presented in figure 4.2.

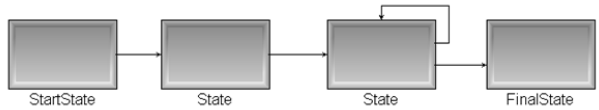


Figure 4.2: A finite-state machine model

4.2 Meta-Modeling Tools Comparison

There are several configurable modeling tools available. This section is a comparison of advantages and disadvantages of the most popular ones.

4.2.1 MetaEdit+

MetaEdit+ is a powerful, commercial application. According to the authors, it is one of the most widely used Domain-Specific Modeling Environment. There are many examples of domain-specific models on the company webpage: from web and smartphone applications to industrial machine controllers and automotive product line feature model. An example of modeling environment for arithmetic logic is presented in figure 4.3. It presents a model of automotive electronics. The model is composed of sensors, logic gates and actuators.

Although MetaEdit+ is very powerful, it has a number of limitations. First of all, it is an expensive tool (costs almost 6000 Euros). The other disadvantage, which definitely eliminates usage of this tool in this thesis, is a lack of documentation of the generated meta-model files, which makes impossible to use the files as an input to our model generator [1].

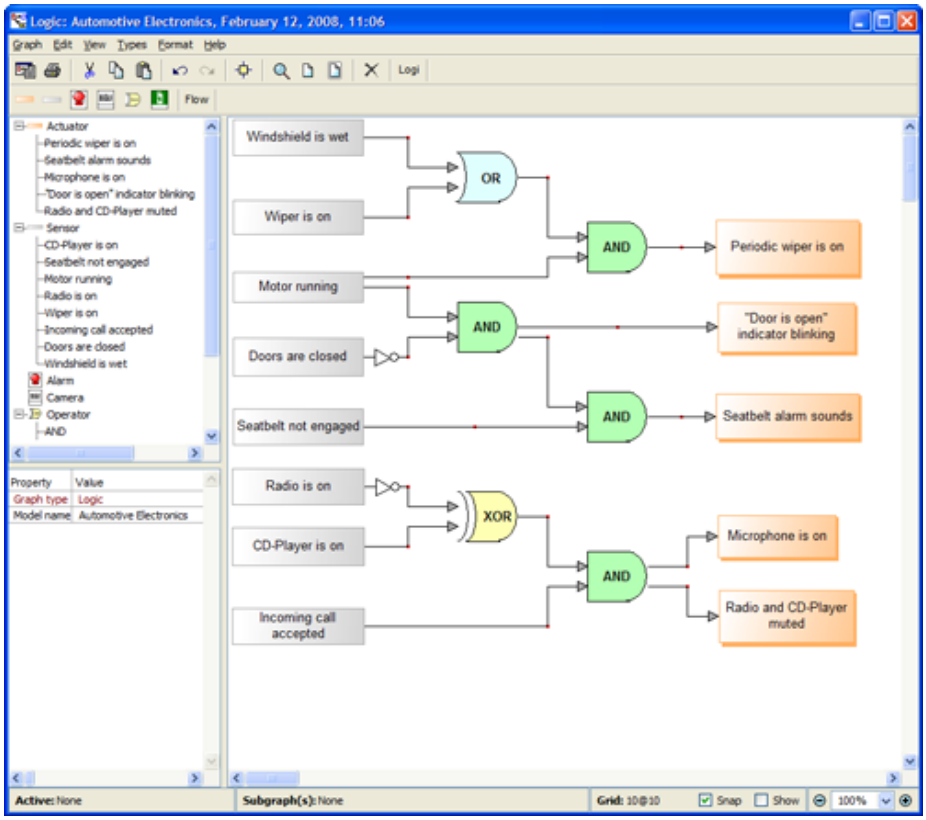


Figure 4.3: MetaEdit+ modeling environment for arithmetic logic. *Source: [1]*

4.2.2 The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a project developed for the Eclipse platform. It was mainly intended to create models of Java applications. It supports automatic Java code generation based on the described model. EMF accepts many ways of describing models: annotated Java, XML files, or different kinds of modeling tools (e.g. Rational Rose). An example of a graphical tool input is presented in figure 4.4. It is a class diagram of an Eclipse component.

Although the Eclipse Modeling Framework is well documented, it has some disadvantages. Firstly, it is mainly focused on program modeling and code generation rather than being a completely generic modeling environment. Secondly, the Eclipse Modeling Framework does not support any mechanisms, which are able to apply advanced constraints to a model [16][2].

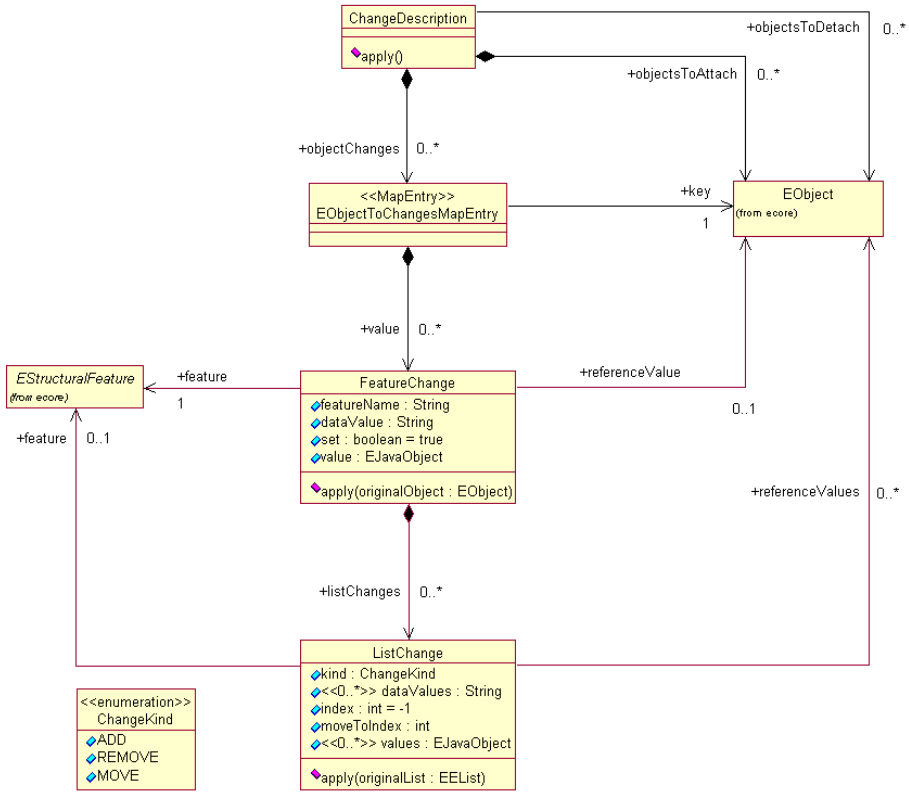


Figure 4.4: A class diagram specified using EMF. *Source: [2]*

4.2.3 The Generic Modeling Environment

The Generic Modeling Environment (GME) is a free academic tool. It is a meta-programmable tool, which may be customized using UML-based class diagrams. Classes may be associated with constraints described in Object Constraint Language (OCL). It is a functional programming language, with structure similar to SML. All constraints are expressed as statements, which return a boolean value. Statements may be composed of conditional `if ... else` statements, they may use local variables declared with `let` expression and they may perform operations on object collections.

After defining a meta-model, it may be registered in GME. After that, a user may create a model-specific environment. This environment contains all entities and their attributes defined in a meta-model. A user is able to create a

model using those entities and he may verify if a model does not violate defined constraints.

As mentioned before, GME comes with a UML-based meta-model that is used to describe a domain-specific environment (in other words, to generate other meta-models). There is a large number of elements that may be used by a model designer. On the top of entities hierarchy, there is a `Project` entity. It contains `Folders`, which help grouping all the objects (similarly to folders in a file system). `Folders` contain `Atoms`, `Models`, `Connections`, `References` and `Sets` (together called as `FCOs`). `Atom` is the most basic component, which cannot contain any other part. `Connections` are used to express relation between entities. It may be directed or undirected and minimum and maximum multiplicity may be specified. `Models` represent more complicated parts of a meta-model. They may be composed of different type of entities (other `FCOs`). Each `Model` component has a specified `Role`, defined by a model creator. Each model has an `Aspect`. `Aspects` specify the visibility of a `Model`. User can choose an aspect, and only the `Models` associated with it are visible. `Reference` is an analogy to a pointer in HLL. It is not a physical object, but just a reference to the specified `FCO`. A `Set` is a group of objects with the same properties. Each `FCO` may be associated with an attribute of integer, double or string type [11][15]. A diagram of GME components hierarchy is presented in figure 4.5.

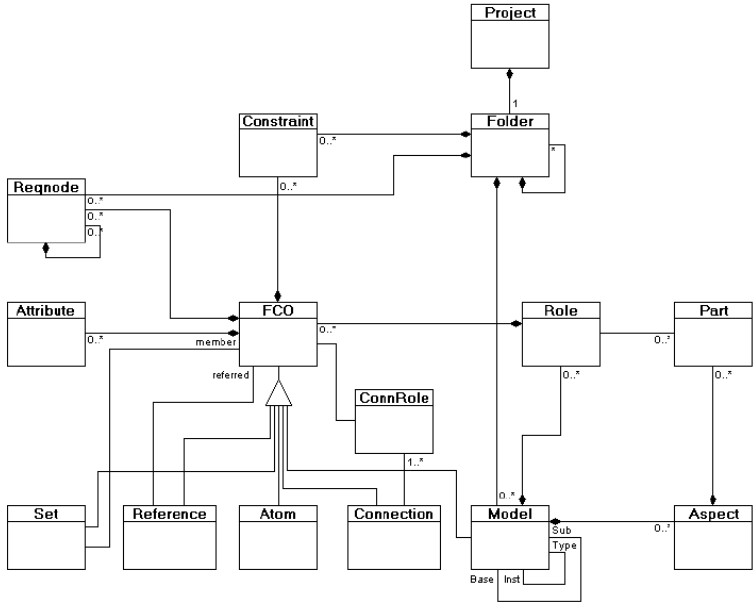


Figure 4.5: GME components hierarchy. *Source: [15]*

Apart from the rich entities collection, GME has other interesting features. First advantage of using Generic Modeling Environment is Object Constraint Language (OCL) support. Many advanced constraints may be easily expressed using it, e.g. maximum/minimum parameter value, maximum number of connections, multiplicity of connections, etc.

Secondly, it is possible to write a GME plug-in. It can be developed in any language supporting COM technology or in Java. Plug-ins have access to all the meta-model components, which means that it is possible to develop a meta-model interpreter. Having such a tool, a user may generate models directly from GME, without any intermediate steps (like generating XML output files and then using them as generator input) [11].

GME seems to be the best program to create the input meta-model for the model generator. To make the model generator tool simple and user friendly, it should be developed as GME plug-in.

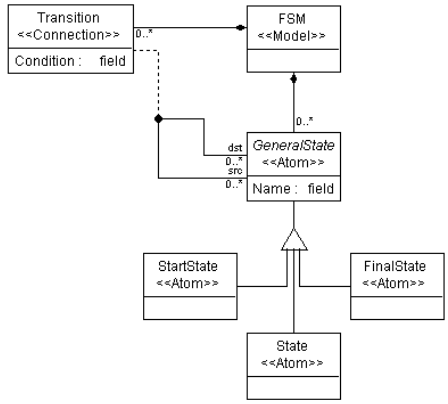
4.3 GME Meta-Model vs. Graph Model as a Meta-Model

After choosing the modeling tool, another question must be answered: is it possible to create a GME interpreter of a native GME meta-model? Or maybe a dedicated meta-model, describing basic model elements would be better to use.

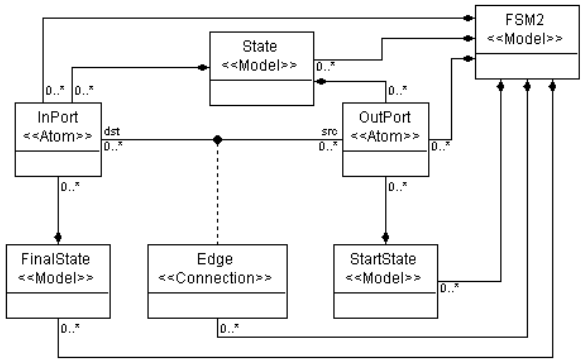
A model may be constructed from many different types of entities. GME supports also `Atoms` and `Models` inheritance. All these elements and OCL constraints allow creating very advanced solutions. However, the problem is that the same meta-model may be described in many different ways. As an example, two meta-models of finite state machine were created. The class diagrams of the meta-models are presented in the figure 4.6.

In the first approach (figure 4.6(a)) inheritance is used. There is an abstract class `GeneralState` which is then inherited by `StartState`, `State` and `FinalState`. Each kind of state is associated with `Name` parameter. States are connected by `Transitions`. `Transitions` have one `Condition` attribute. In order to avoid edges outgoing from the final states and edges incoming to the starting states, two OCL constraints associated with `GeneralState` are used:

```
-- eliminates edges outgoing from the final states
let connFCOs = self.connectedFCOs("src") in
```



(a)



(b)

Figure 4.6: Two different meta-models of finite state machine.

```
  connFCOs->forAll(e | e.kindName <> "FinalState")

-- eliminates edges incoming to the starting states
let connFCOs = self.connectedFCOs("dst") in
  connFCOs->forAll(e | e.kindName <> "StartState")
```

In the second solution presented in figure 4.6(b), states are represented by `Models`. Each kind of state is associated with at least one type of port `Atom`. There are two types of ports: `InPort` (incoming port) and `OutPort` (outgoing port). In order to connect two states, first outgoing port has to be added to the origin and incoming port has to be added to the destination. After that, ports

may be connected with `Edge` connection. `State` is associated with both `InPort` and `OutPort`, `FinalState` may only contain `InPorts`, whereas `StartState` is related only with `OutPorts`. Such a configuration eliminates need of using OCL, `StartState` can never have incoming edges and `FinalState` can never be outgoing edge origin.

A finite state machine is a rather simple model of computation. However, as shown GME allows model creator to describe it in many different ways, using different mechanisms and techniques. Therefore, it is impossible to write an interpreter, which would be able to interpret raw GME meta-model correctly in all cases. Inheritance, hierarchy of objects and OCL constraints give infinite number of possibilities in model descriptions.

In order to describe embedded system models, there is no need to use the full GME meta-model. Our solution is to define entities specific to embedded system models. Our model generator tool will be able to unambiguously interpret meta-models created using the proposed entities.

4.4 Embedded Systems Meta-Model

This section is a description of our proposed entities for describing embedded systems meta-models. There are two main aspects:

- *Vertices*, which is a specification of vertex types, connection rules and model attributes,
- *Architecture*, which is a specification of model structure (number of start vertices, number of graphs and nodes, etc.)

Each meta-model entity is associated with OCL constraints in order to assure the meta-model correctness.

4.4.1 Vertices Aspect

This section describes all the entities belonging to the *Vertices* aspect. Figure 4.7 presents class diagram for all of the entities.

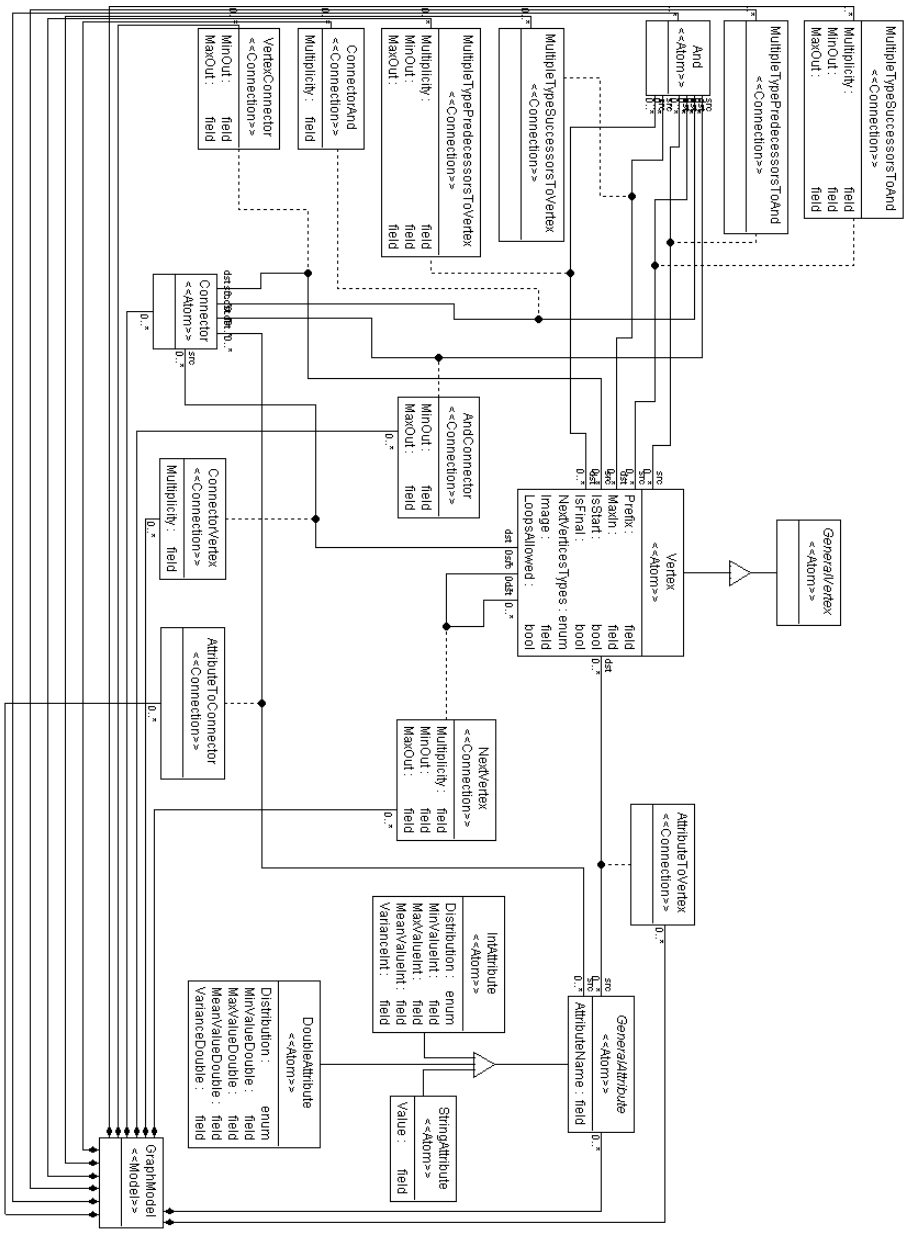


Figure 4.7: Diagram of entities belonging to the *Vertices* aspect

4.4.1.1 Vertex Type

As mentioned in section 2.5, graph is a set of vertices and edges. `Vertex` represents the type of vertices used in a model – directed acyclic graphs are represented using only one `Vertex` entity, whereas Petri nets require two `Vertex` types (Places and Transitions). `Vertex` inherits from abstract `GeneralVertex` `Atom`. The other two classes, which inherit from `GeneralVertex` belong to `Architecture` aspect and they will be described in section 4.4.2. In order to enforce user to create at least one vertex type, the following OCL constraint is used:

```
let vert = atoms(Vertex) in
  if vert->size() = 0 then
    false
  else
    true
  endif
```

There are a number of parameters characterizing vertices. First one – `Prefix` is a string parameter that specifies the prefix name for the vertex type. Vertex name in a model is a concatenation of the prefix and vertex number.

The second parameter is `MaxIn`. It is an integer parameter that specifies maximum number of incoming edges. This parameter cannot be negative, therefore `Vertex` is associated with the following OCL constraint:

```
self.MaxIn >= 1
```

`IsStart` and `IsFinal` are two Boolean parameters. They specify if vertices of the type may be used as the starting (final) ones. E.g., in Petri net Places have both parameters set to true, whereas transitions have both of them set to false. Each model must contain at least one type of vertex, that may be used as the starting vertices and one type that may be used by final vertices. In order to enforce that requirement, two OCL constraints are defined:

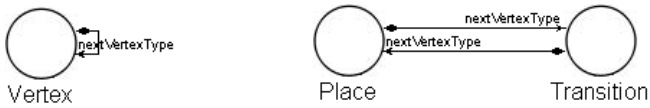
```
-- at least one type may be used by starting vertices
let vert = atoms(Vertex) in
  vert -> exists( v : Vertex | v.IsStart = true)

-- at least one type may be used by final vertices
let vert = atoms(Vertex) in
  vert -> exists( v : Vertex | v.IsFinal = true)
```

`NextVerticesTypes` is a parameter that may have one of two possible values: `OR` or `XOR`. The parameter will be described in details together with the description of `NextVertex Connection`.

`Image` specifies bitmap name, which is used as an icon for the vertices of the associated type. The bitmaps must be placed in *icons* folder, in the same location as the meta-model.

`LoopsAllowed` is a Boolean parameter that specifies if self-loops are allowed (edges with the same origin and destination vertex).

4.4.1.2 Connection Type



(a) (b)

Figure 4.8: The meta-models of (a) directed acyclic graph and (b) Petri net

In order to define which vertex type may be used as the specified type successor, `NextVertex` connection is used. Figure 4.8(a) presents the simplest meta-model of a directed acyclic graph. There is only one vertex type, therefore, all its successors must be of the same type – this is why `NextVertex` connection is a self-loop. Figure 4.8(b) is the simplest meta-model of Petri net. There are two types of vertices – *Place* and *Transition*. Successor of *Place* type vertex may only be *Transition* type vertex. Similarly, *Transition* type vertices are predecessors of *Place* type vertices.

`NextVertex` is a directed edge associated with three parameters. `MinOut` and `MaxOut` specify minimum and maximum number of outgoing edges for the origin vertex type. As an example the Petri net meta-model presented in figure 4.8(b) will be used. If user sets `MinOut` to 2, `MaxOut` to 4 for the connection between *Place* and *Transition* (the upper one), then all *Places* in a model will have from 2 to 4 edges outgoing to different *Transitions*. Multiplicity of connections (number of edges between pair of the same vertices) is specified by the third parameter `Multiplicity`. All three parameters, associated with `NextVertex`, have to be nonnegative. Additionally `Multiplicity` cannot be larger than `MaxOut` parameter and `MinOut` must be smaller or equal to `MaxOut`. In order to enforce this requirement, the following OCL constraints are associated with `NextVertex` connection:

```
-- all the parameters nonnegative
self.Multiplicity >= 1
self.MinOut >= 1
self.MaxOut >= 1

-- MaxOut greater or equal to MinOut
self.MaxOut >= self.MinOut

-- Multiplicity smaller or equal MaxOut
self.Multiplicity <= self.MaxOut
```

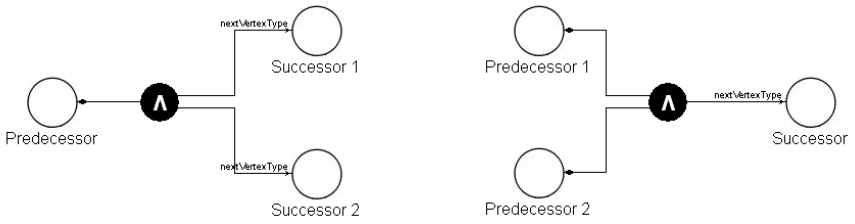
There are also two other constraints, that are related to `NextVertex` connection, but they are associated with `Vertex Atom`. The first one enforces specifying successor type for each vertex type. The second one eliminates multiply `NextVertex` connections between the same two vertex types.

```
-- duplications of the same connections are not allowed
let nextBag = self.bagConnectedFCOs("src", NextVertex) in
  let nextSet = self.connectedFCOs("src", NextVertex) in
    nextBag ->size() = nextSet->size()

-- successor type must be specified
self.connectedFCOs("src", NextVertex)->size() >= 1 or
self.connectedFCOs("src", ConnectorVertex)->size() >= 1
```

Each vertex type may have more than one successor type. However, using multiple types of successors makes the model ambiguous. If a type has 3 different successor types, does it mean that it may have up to three different successors at the same time or only one of them may be used? In order to answer this question `NextVerticesTypes` parameter of `Vertex Atom` is introduced. If it is set to `OR` than any number of successor types may be used, whereas if it is set to `XOR`, only one of them may be used.

4.4.1.3 Multiple Successor or Predecessor Types

In some cases, the model structure is more complex, and hence it is not enough to specify just one successor type. A good example is sequencing diagram for biochips described in section 2.4. It requires more powerful mechanisms to specify multiple predecessor/successor types. For example, predecessors of a mixing node may be: 2 inputs, 2 mixings, 1 mixing and 1 input or 1 dilution and 1 input. To define this type of specification, we have introduced the entity `And Atom`.



(a) (b)

Figure 4.9: Multiple successor/predecessor types

An example of multiple successor types is presented in figure 4.9(a). One vertex type – *Predecessor* has 2 types of successors: *Successor 1* and *Successor 2*. In order to connect predecessor with `And` entity, `MultipleTypeSuccessorsToVertex Connection` is defined. `MultipleTypeSuccessorsToAnd Connection` is used to connect `And` entity with one of the successors. There are three parameters that are associated with `MultipleTypeSuccessorsToAnd Connection`: user can specify minimum and maximum number of successors of the connected successor type as well as maximum multiplicity of the connection.

Multiple predecessor types are defined in a similar manner. An example of such a case is presented in figure 4.9(b). Each predecessor is connected with an `And` entity by a `MultipleTypePredecessorsToVertex Connection`. The user can specify a maximum multiplicity, as well as the minimum and maximum number of predecessors of the connected predecessor type. In order to connect an `And` entity with the successor type, a `MultipleTypePredecessorsToAnd Connection` is defined.

There are five OCL constraints associated with the `And` entity. The first one checks if only one predecessor is defined for multiple successor types. The second constraint checks if there is only one successor for multiple predecessor types. Next two have a similar role, but they are triggered whenever a new connection to `And` entity is added. In order to avoid undesired errors, conditions are relaxed (they accept 0 or 1 predecessor/successor type, whereas first two require exactly one). The last constraint eliminates mixing different sort of connections (`MultipleTypePredecessorsToVertex` with `MultipleTypeSuccessorsToAnd`, `MultipleTypeSuccessorsToVertex` with `MultipleTypePredecessorsToAnd`, etc.). The constraints are presented below.

```
-- only one predecessor type for multiple successor types
let connMultSucAnd =
  self.bagConnectedFCOs(MultipleTypeSuccessorsToAnd) in
  let connMultSucVert =
```

```
    self.bagConnectedFCOs(MultipleTypeSuccessorsToVertex)
        in
    (connMultSucAnd.size() <> 0 implies
      connMultSucVert.size() = 1)

-- only one successor type for multiple predecessor types
let connMultPredAnd =
  self.bagConnectedFCOs(MultipleTypePredecessorsToAnd) in
  let connMultPredVert =
    self.bagConnectedFCOs(
       MultipleTypePredecessorsToVertex) in
    (connMultPredVert.size() <> 0 implies
      connMultPredAnd.size() = 1)

-- triggered version of the first constraint
self.bagConnectedFCOs(MultipleTypeSuccessorsToVertex).
    size() <= 1

-- triggered version of the second constraint
self.bagConnectedFCOs(MultipleTypePredecessorsToAnd).size
    () <= 1

-- eliminates mixing different sort of connections
let connMultPredAnd =
  self.bagConnectedFCOs(MultipleTypePredecessorsToAnd) in
  let connMultSucAnd =
    self.bagConnectedFCOs(MultipleTypeSuccessorsToAnd) in
    let connMultPredVert =
      self.bagConnectedFCOs(
         MultipleTypePredecessorsToVertex) in
      let connMultSucVert =
        self.bagConnectedFCOs(
           MultipleTypeSuccessorsToVertex) in
        (connMultPredAnd.size() <> 0 implies
          connMultSucAnd.size() = 0) and
        (connMultPredAnd.size() <> 0 implies
          connMultSucVert.size() = 0) and
        (connMultSucAnd.size() <> 0 implies
          connMultPredAnd.size() = 0) and
        (connMultSucAnd.size() <> 0 implies
          connMultPredVert.size() = 0)
```

4.4.1.4 Attributes

Each embedded system model contains different attributes. They specify different properties, such as, process execution time, communication cost, number of tokens. Attributes are usually integer or floating point numbers. Some model also use string attributes as different kinds of labels. Our proposed entities contain three type of attributes: an integer attribute may be also used as a Boolean attributes, if the range of values is set from 0 to 1. Each attribute type inherits from abstract `GeneralAttribute Atom`. This entity has one string parameter – `AttributeName`. Attribute name must be unique and cannot be empty. Additionaly every attribute has to be associated with a vertex type or a connector. The following constraints enforce those requirements:

```
-- attribute name cannot be empty
self.AttributeName.trim() <> ""

-- attribute name must be unique
let vertBag = self.bagConnectedFCOs("dst",
    AttributeToVertex) in
  let vertSet = self.connectedFCOs("dst",
      AttributeToVertex) in
    let connBag =
      self.bagConnectedFCOs("dst", AttributeToConnector)
        in
      let connSet =
        self.connectedFCOs("dst", AttributeToConnector)
          in
        vertBag ->size() = vertSet->size() and
        connBag ->size() = connSet->size()

-- attribute must be connected to a vertex type or a
    connector
self.connectedFCOs("dst", AttributeToVertex)->size() >= 1
    or
self.connectedFCOs("dst", AttributeToConnector)->size()
   >=1
```

`StringAttribute Atom` contains one string field – `Value`, which is value of the attribute. It is a fixed value, specified by a user, since random string attributes are rather useless. In order to avoid empty string attributes, the following OCL constraint is associated with `StringAttribute Atom`:

```
self.Value.trim() <> ""
```

Values of `IntAttribute` and `DoubleAttribute` are random in generated models. The user may specify 5 different attribute parameters. The first one is `Distribution`, which may take one of the three values: Uniform, Normal or Exponential. These three types of random number distributions are the most widely used in embedded system models [29]. The user can also specify minimum and maximum value of the random attributes. Last two parameters – mean value and variance are required only for some distributions (mean value for exponential and normal distribution and variance for normal distribution).

There are three OCL constraints associated with `IntAttribute` and `DoubleAttribute`. The first one enforces the maximum value to be greater or equal to the minimum value. The second one is activated only for normal and exponential distribution, when it checks if the mean value is in the range [*min_value*, *max_value*]. The last constraint is checked only if normal distribution is chosen. It checks if the variance is nonnegative. The code of the constraints is presented below.

```
-- minimum value must be less or equal to the maximum
    value
self.MaxValueInt >= self.MinValueInt

-- for normal and exponential distribution,
-- mean value must be in range [min; max]
if (self.Distribution = #Normal or
  self.Distribution = #Exponential) then
    self.MeanValueInt >= self.MinValueInt and
    self.MeanValueInt <= self.MaxValueInt
else
  true
endif

-- for normal distribution, variance must be nonnegative
if (self.Distribution = #Normal) then
  self.VarianceInt >= 0
else
  true
endif
```

Attributes may be associated with either vertices or edges. In order to associate an attribute with a vertex type, `AttributeToVertex Connection` is created. Attributes and edges association is slightly more complicated. The most obvious solution is to create connection between `NextVertex Connection` and `GeneralAttribute Atom`. However, it is impossible to associate an entity with a connection in GME. To achieve the same effect, a new `Atom`, namely

`Connector`, is introduced.

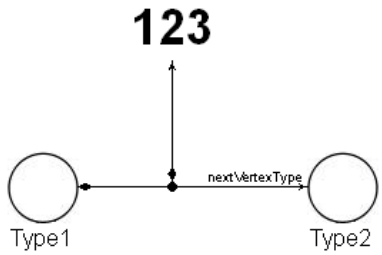


Figure 4.10: Integer paramiter associate with edges family

An example using `Connector` is presented in figure 4.10. In order to associate an attribute with the edges family from *Type 1* vertices to *Type 2* vertices `Connector` is added (small, black dot between two vertex types). It is then connected to the both vertex types and to the integer parameter. `Connector Atom` is associated with four different connection types: `ConnectorVertex` and `VertexConnector` that are used for a single successor type, as well as `And-Connector` and `ConnectorAnd` that are used to connect a vertex type with an `And Atom` in case of multiple successors or predecessors.

Each `Connector` can have only one source, one destination and a number of associated attributes. To enforce that requirement, the following OCL constraints are associated with the atom:

```
-- each connector has one source
let vBag = self.bagConnectedFCOs(VertexConnector) in
  let aBag = self.bagConnectedFCOs(AndConnector) in
    vBag.size() + aBag.size() = 1

-- each connector has one destination
let vBag = self.bagConnectedFCOs(ConnectorVertex) in
  let aBag = self.bagConnectedFCOs(ConnectorAnd) in
    vBag.size() + aBag.size() = 1
```

4.4.2 Architecture Aspect

All the entities described so far belong to the *Vertices* aspect. The *Architecture* aspect is focused on the structure of the generated model. It allows the user to specify the number of start vertices, number of model nodes and number of

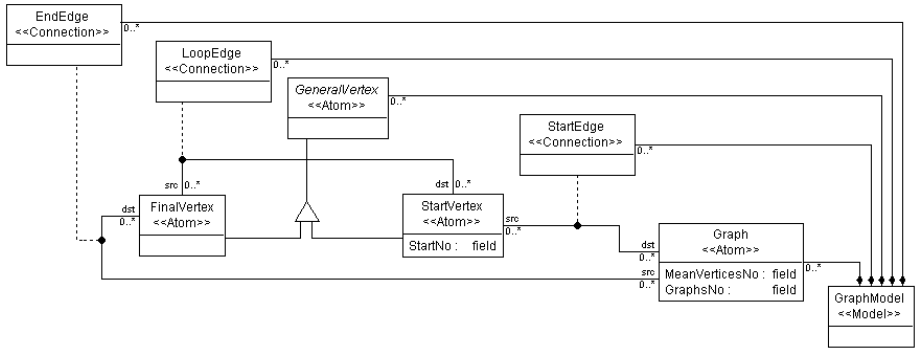


Figure 4.11: Diagram of entities belonging to *Architecture* aspect

models to be generated. The diagram of the entities belonging to this aspect is presented in figure 4.11.

`StartVertex` and `FinalVertex` are two entities that inherit from GeneralVertex (similarly to `Vertex` entity described in section 4.4.1.1). They represent dummy start and final vertices. `StartVertex` has `StartNo` integer attribute, that specifies number of starting vertices. If this parameter is set to 0, number of start vertices is random.

`Graph` is an `Atom` that represents the model structure. The user can specify two parameters: `MeanVerticesNo`, which is a mean number of vertices per generated graph and `GraphsNo` that specifies number of graphs to be generated. `Graph Atom` is connected with `StartVertex` and `FinalVertex` by respectively: `StartEdge` and `FinalEdge`. `LoopEdge` is a connection from `FinalVertex` to `StartVertex`. If it is used, than generated graphs are cyclic, otherwise, acyclic graphs are generated.

The user has to specify one dummy starting and final state. Also, the `Graph` entity has to be defined and connected to starting and final state. To enforce this requirement, the following OCL constraints are associated with the meta-model:

```
-- single starting vertex
let verts = atoms(StartVertex) in
  vert -> size() = 1

-- single starting vertex
let verts = atoms(FinalVertex) in
  vert -> size() = 1
```

```
-- single graph description
atoms(Graph) -> size() = 1

-- single StartEdge (associated with StartVertex)
self.connectedFCOs("dst", StartEdge)->size() = 1

-- single FinalEdge (associated with StartVertex)
self.connectedFCOs("dst", FinalEdge)->size() = 1
```

Usage of the the described meta-model, is presented in chapter 6, using several case studies.

Chapter 5

Model Generator

An embedded system meta-model specified using the entities proposed in the previous chapter is used as the input to the model generator. The meta-model is created graphically using GME as described in the previous chapter. This graphical specification is then used to generate a GME model with a random structure and random values of attributes, according to the given specification. The generator is able to create two other representations of the models: in DOT format and PNG format (using *Graphviz* tool). It is also transformed by the meta-model interpreter into GME meta-model of the specified model. In this way, the meta-model is used not only to generate synthetic models, but also to create a GME-based modeling environment. The modeling environment may be used to create the specified models by hand, or it can load and manipulate the generated synthetic models.

The diagram of the entire process is presented in chapter 1, in figure 1.2. The previous chapter described the first step of the process – meta-model specification using our proposed entities. The next step, which is construction of graphical representation of a specific model, is performed graphically by the user in the GME environment. This chapter presents the details of the last steps performed by GME plug-in. Section 5.1 is an overview of the Model Generator, describes tools used to develop it and an overall architecture of the application. Section 5.2 presents details of the meta-model interpreter. Section 5.3 contains details of the model generator, that produces models in GME, DOT and PNG format.

Last section describes how the Model Generator has been integrated with the GME environment, as well as the GUI of the tool.

5.1 Model Generator Overview

As mentioned in section 4.2.3 GME can be extended by using plug-ins developed in any language supporting COM or Java. Plug-ins have direct access to all GME model entities. The whole model structure is mapped into objects and passed to the plug-in. This GME feature is very useful when creating a model interpreter. A user does not have to export the textual representation of a model or perform any intermediate steps. An interpreter may be run directly from the GME GUI, analyze the current model and perform all the required tasks. Interpreting Model Generator as a GME plug-in seems to be the best solution.

The Model Generator is developed in Java (JDK 6). The Java language has been chosen, because it is a modern, object oriented language. Java programs are more error-free than C++ applications. Automatic garbage collection protects from memory leakage. Java arrays cannot be accessed outside theirs range. JDK comes with huge amount of built-in libraries, support for many different types of collections and other useful tools (like *javadoc*, used for automatic documentation generation). Apart from these advantages, Java programs are nowadays almost as fast as C++ programs. What is also important, is that there are a lot of different, free programming environments for Java.

The source code of the application is almost 6000 lines long. There are 22 classes defined. The code is divided into 4 packages. The first one, `interpreter`, contains classes related to the graph meta-model's interpreter. `Graphgenerator` package contains all the classes that are used during generating of synthetic models. As mentioned in section 4.4.1.4, describing model attributes, there are 3 most widely used random attributes distributions: normal, exponential and uniform. JDK contains only uniformly distributed random numbers generator. The third application's package, `random`, contains implementation of all three kinds of random number generators. The last one, `gui` package, contains forms and other classes related to the graphical user interface. The class diagram of the entire application is presented in appendix B.

All program features have been tested using several case studies. They are described in details in chapter 6.

5.2 Meta-Model Interpreter

This section provides the details of the meta-model interpreter, which takes as input meta-models constructed using the entities presented in chapter 4. The interpreter is responsible for extracting all the information about the embedded application meta-model: vertex types, associated attributes and all connection rules specified by a user. The data is then used to generate synthetic models based on the input meta-model and to convert the meta-model to an XML format compatible with GME. The user can register a meta-model in GME and then open and modify generated GME models. The XML Schema, which defines format of XML document with meta-model description is presented in appendix A.

The core class of the interpreter is `GMEInterpreter` class. It implements `BONComponent` interface, which contains one function: `invokeEx(JBuilder builder, JBuilderObject focus, Collection selected, int param)`. The function is called by GME when a plug-in is run. It creates the builder object network, which is a collection of Java objects representing `Folders`, `Atoms`, `Models`, `Connections`, `References` and `Sets` [10].

A builder object network is not really a desired input for our Model Generator. For example, in order to find attributes associated with a connection type, first a connection between origin vertex type and connector must be found, then connection from connector to a destination vertex type and finally connection from a connector to an attribute. Moreover, vertex type and attributes are mapped to the same type — they are both represented as `Atoms`.

Instead of using a builder object network, the meta-model interpreter transforms it into a different data structure, dedicated to our solution. It has a separated representation for the following entity types:

- attributes (string, integer or double)
- vertex types
- connection types (connecting a pair of vertex types or multiple successor/predecessor types)

The next sections describe the details of the mentioned data structure.

5.2.1 Attribute Representation

There are three types of attributes: string, integer and double. Each attribute has a name and unique ID. The ID is used during meta-model generation in XML format, where each meta-model element is associated with its own identifier. An XML representation of an attribute is very simple. It is one element `attrdef`, with four attributes: `name`, which is attribute name, `metaref` – a unique ID, `defvalue` that specifies default attribute value and `valuetype`, which is attribute type name (integer, double or string).

In order to create the most general attribute representation abstract class `Attribute` is part of `interpreter` package. It contains name field and `toMetaXML` abstract function declaration. The function is supposed to modify a passed Document Object Model (DOM) object, containing an XML meta-model, by adding attributes description to associated connections or atoms.

The class is inherited by three derived classes, which are concrete type attributes representation. `StringAttribute` class represents a string attribute. In the created graph meta-model, apart from a name, only value is specified for this kind of attribute. To keep this information, `value` field is created.

Two other derived classes – `IntegerAttribute` and `DoubleAttribute` are very similar to each other. They provide the same functions and keep the same data, but the first one uses integer data type, whereas the second one uses floating point number representation. They keep distribution type (flags for each of the used distributions, are defined in `Attribute` class, maximum, minimum and mean attribute value, as well as attribute variance. Both classes implement also `getValue` function, which returns random attribute value, depending on the specified parameters. Value is generated by the suitable random number generator from the `random` package. The random value generation process is described in details in section 5.3, where model generator, using `getValue` function, is described.

5.2.2 Vertex Type Representation

Vertex type is a meta-model element which has some common properties with connection types. They both have names and unique IDs. Moreover, they may both be associated with different attributes. These common features of vertex and connection types are implemented in `VCType` class. It contains `name` and `ID` fields, as well as three lists, one for each attribute type. Apart from field accessors, `addAttributes` method is implemented. It adds attributes

associated with a vertex or a connection type. The attributes are obtained from `JBuilderObject Vector`, which is created by `GMEInterpreter` class. The method uses `id` argument, to identify the connector related to a connection type or the atom related to a vertex type. Only attributes connected to the specified id are added to the appropriate list.

`VertexType` is one of the two classes that inherit from `VCType` class. It represents a vertex type, therefore it has `prefix, image, maxIn, isFinal, isStart, loopsAllowed` and `nextTypes` fields, to keep meta-model `Vertex Atoms` attributes related. Similarly to attributes, `VertexType` also implements `toMetaXML` function. It creates a DOM part, representing a vertex type. It is represented as an `Atom`. User gets a separate entity for each vertex type. To describe them in XML format, small sub-trees are created. A root element of an atom description is `atom` element. It has three attributes: `name`, `metaref`, which is entity's unique id and `attributes`, which is a list of comma-separated attributes names. With the last XML attribute, all the string, integer and double attributes, defined in meta-model and described in `attrdef` element, are associated with a particular vertex type.There are two mandatory children of `atom` element – two `regnode` elements. They both have `name` and `value` attributes. The first `regnode` element specifies icon used for a vertex type. The `name` attribute is set to *icon*, whereas `value` attribute is a bitmap name. The second `regnode` element is a flag specifying if a vertex type name should be visible. The `name` attribute is set to *namePosition* and the `value` attribute is 0 or 1. Each `atom` element may also have any number of `constraint` element children, which define constraints associated with a vertex type. The details of constraints added to vertex types are described in section 5.2.5.

In order to find vertex type successors and predecessors two list are added to `VertexType` class. They both contain `ConnectionType` class objects. The details about this class are given in the next paragraph.

5.2.3 Connection Type Representation

`ConnectionType` is the second class, which inherits from `VCType` class. It represents a connection between two different vertex types – it specifies successor/predecessor type. In order to keep meta-model information about multiplicity, minimum and maximum number of edges, it contains three fields: `multiplicity`, `minOut`, `maxOut`. It also contains two fields, specifying origin type and destination vertex type (`from` and `to`). The class is supposed to reflect `NextVertex Connection`, but also all connections related to multiple successor and predecessor types. GME requires different `Connections` entity for each pair of `Atom` types, which means that vertex with e.g. three successor types requires three dif-

ferent `Connection` entities. However, `ConnectionType` has to keep information about other `ConnectionTypes` that lead to a different successor or predecessor type. This information allows generating synthetic models exactly according to a user specification. If other successor types were not kept, then only random successor types number would be connected, since generator would treat them as completely unrelated entities. In order to store information about other successor or predecessor type, `ConnectionType` class contains two `ConnectionType` type list – `otherSuc` and `otherPred`.

`ConnectionType` also overrides `toMetaXML` function. In XML representation of GME meta-model, all `Connection` entities have `connection` root element. Its attributes specify name, unique id and all constraints associated with a type. There are five `connection` element children: four `regnode` elements and `connjoint` element. Attributes of `Regnode` element specify color, line style, and ending styles (since it represents directed graphs edge, one end is solid, whereas the second one is arrow). `Connjoint` element have two children specifying source and destination atom types.

5.2.4 Interpreting Process

As mentioned before, `GMEInterpreter` class is a core class of the meta-model interpreter. It keeps collections of all meta-model vertex and connection types. The other information stored by the class objects are: meta-model name, mean number of vertices per graph, number of graphs, number of starting nodes and flag specifying if cycles are allowed. All this information is extracted from builder object network, generated by `invokeEx` function. The interpreting process is performed in three iterations through the object network.

In the first step, all the model vertices are found. All vertex type parameters are read from atom properties and stored in a `VertexType` class object. Also all the parameters associated with a type of vertex are found by `addAttributes` (described in section 5.2.2). Finally, created `VertexType` class object is added to a collection of all vertex types in a meta-model.

During the second iteration all connection types are found. The process is much more complicated than the one performed in the first iteration, since there are different `Connection` entities used for single and multiple successor/predecessor types. If connection type is `NextVertex` then all its parameters are read and stored in `ConnectionType` class object. After that origin and destination vertex types are added and finally the object is added to the collection of all connection types.

If there is an attribute added to a connection type, then `ConnectorVertex` and `VertexConnector` connections are used. In such a case first `ConnectorVertex` is found, then, by finding outgoing edges from a `Connector`, destination vertex type is found and all the associated attributes are added. All this information is stored in `ConnectionType` class object, which is finally added to the collection of all connection types.

In order to find connection type with multiple successor and predecessor types, two similar actions are performed. For that reason, only finding multiple predecessor types will be described. Firstly, a connection between successor and `And` entity is found. Afterwords, all outgoing connections are found. They may lead directly to one of the predecessors or to a `Connector` (when there is an attribute associated with a connection). Each of the connections, from a successor to one of its predecessors, is added separately, using dedicated `ConnectionType` class object (as described in two previous paragraphs). After finding all predecessor types, information about other predecessors is added to each relevant `ConnectionType` class object.

The last iteration through the builder object network extracts information about graph architecture. Number of starting vertices is read from `StartVertex` entity attribute. Number of graphs and mean number of vertices per graph is extracted from `Graph` entity. If `LoopEdge` entity is found in builder object network, then `cyclesAllowed` flag becomes true. Otherwise, it is set to false.

After generating all objects representing meta-model of an embedded application model, the program can generate XML representation of the meta-model. In order to perform this operation, `toMetaXML` is implemented. It creates root element – `paradigm`, root folder and root model. All the meta-model entities are added calling `toMetaXML` functions declared in `Attribute` and `VCType` classes. They are all associated with one, default aspect, namely `ProjectAspect`.

5.2.5 Constraints Generator

In order to verify, if synthetic models are generated according to a meta-model creator intensions, some OCL constraints are added to a generated meta-model. All constraints are generated by `ConstraintsGenerator` class, added during meta-model interpreting.

The following constraints are generated:

- number of edges outgoing from vertices of a type that cannot be used for

final vertices must be greater than zero;

- number of edges incoming to vertices of a type that cannot be used for starting vertices must be greater than zero;
- attributes must have value from a range $[min, max]$;
- number of edges incoming to a vertex cannot be greater than vertex type `maxIn` property;
- number of edges outgoing from a vertex type, must be from range $[minOut, maxOut]$ specified in a related connection type;
- if `nextType` flag is set to *XOR*, then only one connection type may be used for edges outgoing from a vertex type;
- multiplicity of an edge must be less or equal to a relevant connection type `multiplicity` parameter;
- when a connection type specifies more than one vertex type successor/predecessor, then all required connections, to other successor/predecessor types, also have to go out/come in to a vertex type.

5.3 Model Generator

After the meta-model interpreting, which has built the described data structure, we use the Model Generator to generate the synthetic models. There are three classes related to this process: `Graph`, `Vertex` and `Connection`. `Vertex` class represents each model node. It contains a unique name, list of incoming and outgoing edges, as well as random attributes values. `Connection` class represents each model edge. It contains references to origin and destination vertex and attributes values. Both classes implement `toXML` function, which creates XML vertex/connection representation, compatible with GME.

`Graph` class represents an entire model. It stores list of all model vertices and connections. The class implements `toXML`, `toDOT` and `generateRandom` functions. The first one creates XML representation of a model. Having meta-model in XML representation, user can register it in GME and import models saved in XML format. They can be later viewed and edited in GME environment. `ToDOT` function creates model representation in `dot` format. Using *Graphviz* tool this textual representation can be transformed into `png` graphical format. `GenerateRandom` function generates a synthetic model with random attribute values and structure. The details of this process are described in the next four sections. The first one describes model structure generating details. The next

three sections describe generating random attribute values with different distribution kinds.

5.3.1 Synthetic Model Generation Algorithm

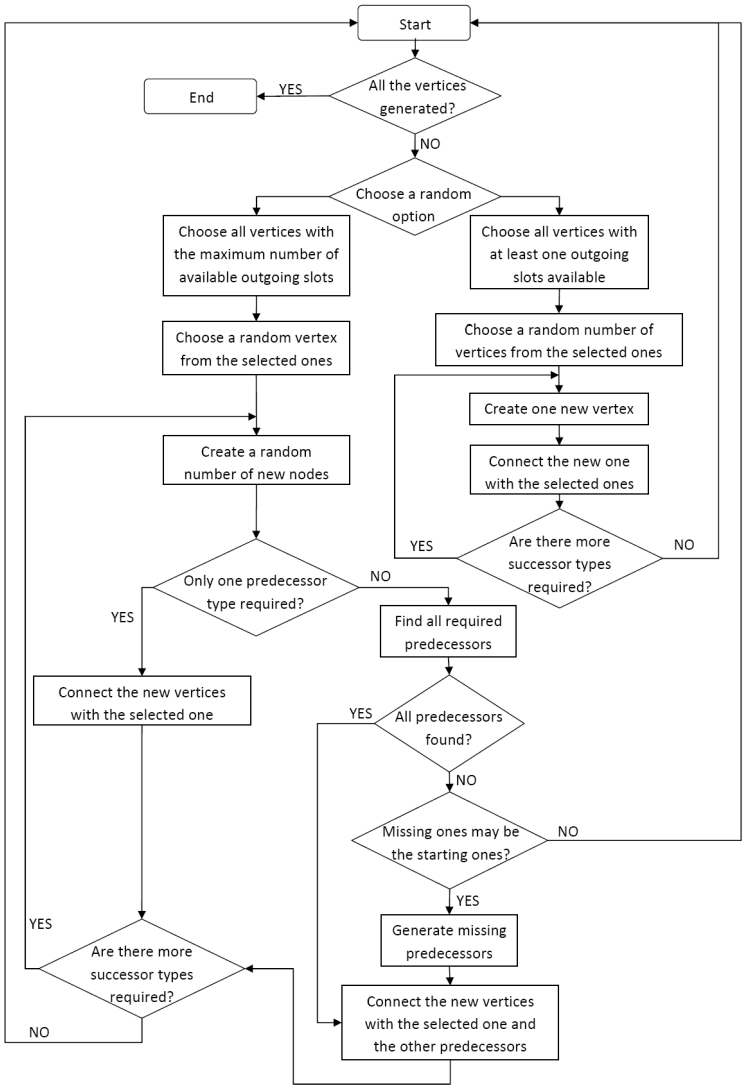


Figure 5.1: The flowchart representing the synthetic model generation algorithm.

As mentioned before, `Graph` class implements `generateRandom` function, which creates synthetic model according to a user specification. The algorithm is quite complex, since model may contain a number of different vertex and connection types, not all vertex types may be used by final and starting vertices, as well as multiple successor or predecessor types may also be defined. The flowchart representing the algorithm is presented in figure 5.1.

The first step of the algorithm chooses randomly one of the two, equally probable options: either the list of vertices, with the maximum number of available outgoing slots is found or the list of vertices with at least one outgoing slot available is created. The next three paragraphs describe steps performed, when the first option is chosen. The steps performed after choosing the second option, are described later in this section. The algorithm steps are illustrated with generation process of a directed acyclic graph model, with maximum 2 edges incoming to a node and with 1 to 2 edges outgoing from a node. There is only one vertex type and one connection type defined. The graphical representation of the directed acyclic graph's meta-model is presented in picture 5.2.

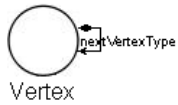


Figure 5.2: A meta-model of a directed acyclic graph, which is used to illustrate the algorithm's steps.

As mentioned before, when the first option is chosen, the list of vertices, with the maximum number of available outgoing slots is found. It is done by checking all connection types that use vertex type as an origin type. In order to avoid favoring connection types with higher maximum number of outgoing edges, proportion of available number of outgoing edges to maximum number of outgoing edges is taken into account. With such an approach, vertex with a maximum of 2 slots and only 1 available slot is chosen, as well as the one with a maximum of 4 slots and 2 of them available, whereas vertex with a maximum of 10 slots and 4 available slots is not added to the list of vertices with the maximum number of available outgoing slots. The exemplary model is presented in figure 5.3. The selected nodes are marked with a gray color.

After creating the list, a random vertex is picked out and the connection type with a maximum number of available outgoing slot is extracted. If there is more than one connection type with the same proportion of available slots to the maximum number of outgoing slots, then a random one is chosen. The actions performed in this step are presented in picture 5.4.

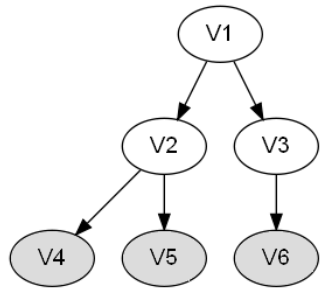


Figure 5.3: Choosing all vertices with the maximum number of available outgoing slots.

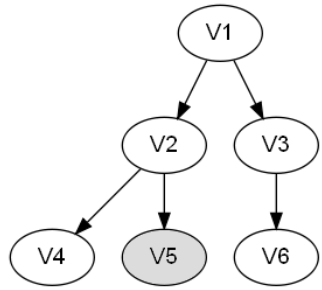


Figure 5.4: Choosing a random vertex with the maximum number of available outgoing slots.

The next steps are repeated for each successor type, defined in a chosen connection type. Firstly, a random number of new nodes is created. The number is drawn from range $[minOut, maxOut]$. Then number of required predecessor types is checked. If there is only one predecessor type, the newly created vertices are connected to the origin (the vertex selected in the previous step) with one or more edges. The number of edges is drawn from from range $[1, multiplicity]$. If there is more than one predecessor type defined for the chosen connection type, firstly all required predecessors are searched for. If there are not enough vertices with available outgoing slots and their type may be used by starting vertices, missing vertices are generated. If finding or generating the entire required predecessor is impossible, then the algorithm gives up and continues from the beginning. Otherwise, it generates edges from the selected vertex and all related predecessors to the newly created vertices and add them to the graph structure.

The exemplary model, after performing the described steps, is presented in figure 5.5.

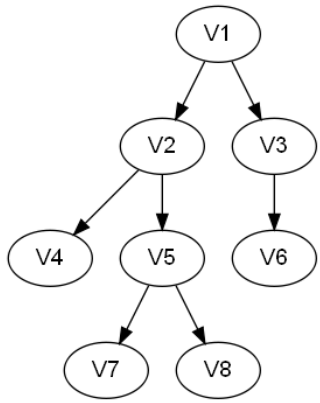


Figure 5.5: DAG model after adding two new vertices.

If the second option is chosen in the first algorithm's step, a different process is performed. First, lists of all vertices with available slots are created. They are stored in a map data structure, with related `ConnectionType` object used as a key. If a connection type requires more than one predecessor types, then it is checked whether the other predecessors are available. If they don't, the vertex is removed from the list. When the lists of all vertices with available slots are created, a random connection type and associated list are picked out and one new vertex is created. The vertex type is the same as destination type of the chosen connection type. This step of algorithm is presented in figure 5.6.

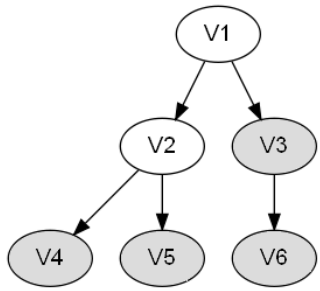


Figure 5.6: Choosing all vertices with at least one outgoing slot available.

From the list of vertices with available outgoing slots, a random number of vertices are chosen. The number of selected nodes is from range $[1, maxIn]$. The result of performing this step is presented in figure 5.7.

In the next step a single node is created. It has the type specified by the

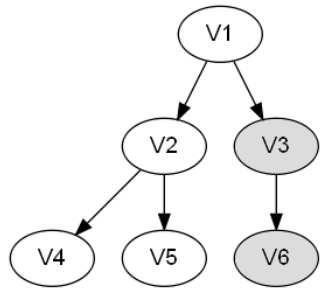


Figure 5.7: Choosing a random number of vertices from the ones with at least one outgoing slot available.

destination type of the connection associated with the selected vertices. These vertices are connected with edges to the newly created vertex. The exemplary model after performing this step is presented in figure 5.8. When this step is completed, the entire process starts from the other successor types.

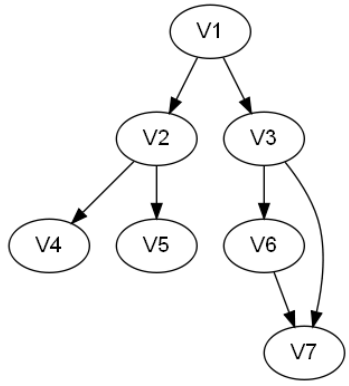


Figure 5.8: The model after adding a new vertex.

When the algorithm finishes a graph generation process, it is possible that nodes, which cannot be used as final ones, have no outgoing edges. This is fixed by `eliminateNotFinals` function. The flowchart of the process for eliminating non-final vertices without outgoing edges is presented in figure 5.9.

In the first step, it creates a list of vertices, which do not have any outgoing edge and cannot be final ones. For each element of the list, connection type leading to a final type vertex is used. If there is no final successor type, then random connection type, which has no other predecessors and leads to a different vertex type, is chosen. If there is still no connection type found, a random one, which

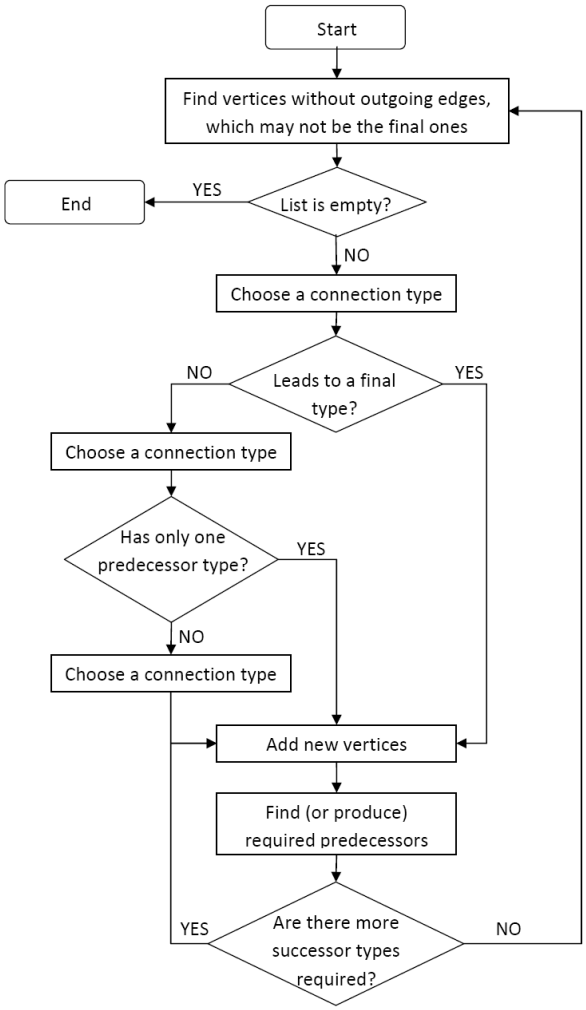


Figure 5.9: The flowchart of the process for eliminating non-final vertices without outgoing edges.

leads to a different vertex type, is picked out.

After finding the connection type, for each successor type, the following steps are performed. Firstly, random number from range $[minOut, maxOut]$ is chosen and this vertices amount is created and connected to a non-final vertex. Then, specified predecessors are either generated (if they may be used as starting vertices), or found in the list of all graph vertices. The process is repeated for

all successor types.

The whole elimination process is performed until there are no non-final vertices without outgoing edges. The generated graph is acyclic, but it may be the case that model allows self-loops or cycles. In the beginning, self-loops are added to a model. Each graph node is checked if its vertex type allows self-loops. If this is the case, an edge which is both starting and leading to the same node is created, with probability equal to 0.5.

If cycles are allowed, `addCycles` function is executed, which is the last step of the generation algorithm. For each final vertex, an outgoing edge is added with probability equal to 0.5. In order to do that, firstly each connection type outgoing from the chosen vertex is found. Afterwords, *minOut* number of vertices with available input slots are found. If there are not enough successors, algorithm gives up. Otherwise, it creates edges from the chosen vertex to all selected successors.

5.3.2 Uniformly Distributed Attributes Generator

Standard Java API contains a random number generator, that produce uniformly distributed numbers belonging to range $[0, 1]$ (or $[0, n)$). Having such a generator, it is easy to generate uniformly distributed numbers with values from range $[min, max]$. In order to do that, values from range $[0, 1]$ (denotated as U_i) have to be linearly transformed using the following formula [22]:

$$V_i = min + (max - min)U_i \tag{5.1}$$

This transformation is performed by `UniformGenerator`, that produce uniformly distributed integer and floating point numbers belonging to range $[min, max]$, using standard Java `Random` generator from `java.util` package.

5.3.3 Normally Distributed Attributes Generator

The second type of random numbers distribution, which is used in synthetic models, is a normal distribution. Standard Java generator produces normally distributed numbers with a mean value of 0 and a standard deviation equal to 1 (standard normal numbers). In order to obtain random values with different

means or standard deviations, standardizing formula is used. A normal distribution is transformed to a standard normal distribution using the following formula [21]:

$$N(0,1) = \frac{N(\mu, \sigma^2) - \mu}{\sigma} \tag{5.2}$$

The formula can be easily transformed, in order to obtain normally distributed numbers with any standard deviation and mean values, using standard normal variables:

$$N(\mu, \sigma^2) = \sigma N(0,1) + \mu \tag{5.3}$$

Normally distributed integer and floating point numbers are produced by `NormalGenerator` class. The class uses `nextGaussian` function from `Random` generator, producing standard normal numbers, and performs the transformation described in this section.

5.3.4 Exponentially Distributed Attributes Generator

The last distribution type, which is widely used during synthetic model attributes generation, is exponential distribution. Java does not provide any function to produce this type of random variables. In exponential distribution probability P that X_i falls in range $[0, z]$ is described by the following equation [22]:

$$P[0 < X_i < z] = \int_0^z p(x)dx \tag{5.4}$$

In the formula $p(x)$ – the probability density function is expressed by the formula:

$$p(x) = \frac{1}{\mu} e^{-x/\mu} \tag{5.5}$$

where μ is distribution mean value. By solving the equation, the following result is obtained:

$$P[0 < X_i < z] = \int_0^z \frac{1}{\mu} e^{-x/\mu} dx = 1 - e^{-z/\mu} \tag{5.6}$$

Probability P is a value from range $[0, 1]$. Having uniformly distributed number U_i from range $[0, 1]$, the following formula is valid:

$$U_i = 1 - e^{X_i/\mu} \Rightarrow X_i = -\mu \ln(U_i - 1) = -\mu \ln(U_i') \tag{5.7}$$

where U_i' is:

$$U_i' = U_i - 1 \tag{5.8}$$

If U_i is uniformly distributed in range $[0, 1]$, then U_i is also uniformly distributed in this range.

Exponentially distributed random numbers generator is implemented in `ExponentialGenerator` class. It uses `nextDouble` function, implemented by standard random Java generator, in order to generate uniformly distributed number in range $[0, 1]$. The value is then transformed using formula 5.7 into an exponentially distributed random variable.

5.4 Using the Tool

As mentioned before, the interpreter and the Model Generator is not a standalone application, but it is implemented as GME plug-in. When user tries to run the program's `jar` file, error window appears with information that the tool has to be run from GME.

The tool comes with `graph.mga` file, which is a graph meta-model described in the previous chapter. Firstly, the meta-model has to be registered in GME environment. In order to do that, user has to open the file with GME and run `MetaGME Interpreter`, located on the application toolbar. All the settings that user is asked about during the registration process, should be set to default values. After meta-model registration, `graph.mga` and all other files, generated

during registration, cannot be moved. It is also important to keep `icons` folder in the same directory. The folder contains bitmaps used to represent meta-model entities. When meta-model is moved to a different directory, registration process has to be repeated, in order to override the old settings.

After registering the graph meta-model, Model Generator has to be added to the list of GME components. In order to do that, *JavaCompRegister* tool is used. It comes with GME and it is located in *%GME_Root_folder%/SDK/Java/* directory. All six program fields have to be filled out. *Name* is the tool identifier, *Description* is the short tool description, *Menu/Tooltip* is a label that appears in GME menu. These three values may be arbitrary chosen by a user. *ClassPath* is a path to the program `jar` file. It has to be a full path, ending with the `jar` file name (by default `GraphGenerator.jar`). *Class* is the interpreter's full class name, including package name – for Model Interpreter it is `interpreter.GMEInterpreter`. *Paradigms* option should remain set to `*`. After registering component in the system, it has to be activated in GME. In order to do that, user has to run GME and choose *Register Components* from *File* menu. In the new window that appears, user has to choose the meta-model interpreter name and press *Toggle* button. After that, a new icon with "Java cup" symbol appears in the GME menu.

Having the meta-model and its interpreter registered, a new meta-model of an embedded system model may be created. It is done by creating a new GME project using `graph` paradigm. The project may be located in any folder. After project creation, a root model must be added to the *Root Folder* (it is done by right-click on *Root Folder*, choosing *Insert Model* option and then *GraphModel* item). The default name of the root model – *NewGraphModel*, should be modified, since it is used as the generated meta-model's name. Then the user can specify the model structure by adding vertex types, attributes and specifying connections between vertex types. Some of the constraints are checked in real-time, during a meta-model creation. However, for some constraints, it is impossible to choose an appropriate trigger (event firing validation). In order to check these properties, user has to validate meta-model, when the design is finished. It is done by choosing *File* menu option and then *Check All* option from *Check* submenu. The user can also display all constraints by choosing *Display Constraints* option from *File* menu.

After validation and eventual mistake corrections, meta-model interpreter may be run. It is done by pressing the button with "Java cup", from GME menu.

The main program window is presented in figure 5.10. On the right hand-side there is a tree of all interpreted meta-model's elements. User can see all vertex and connection types, associated attributes, as well as the architecture details. On the left hand-side user can specify, what should be generated. Checking the

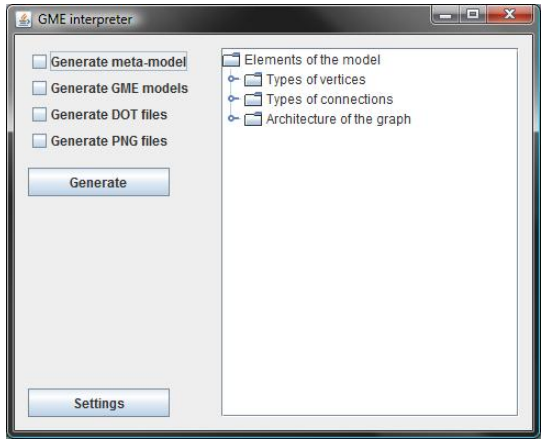


Figure 5.10: Main program window.

first option causes generating GME meta-model in XML format. Checking the second option causes generating models with random structure and saving them in XML format, compatible with GME. In order to open generated models in GME, user at first has to create a new GME project and register the meta-model, generated by the tool, by pressing *Add from file* button. After successful project creation, generated models may be imported using *Import XML* option from *File* GME menu. User can also generate *DOT* file, with a random model description and graphical graph representation in *PNG* format (requires installing *Graphiz* tool).

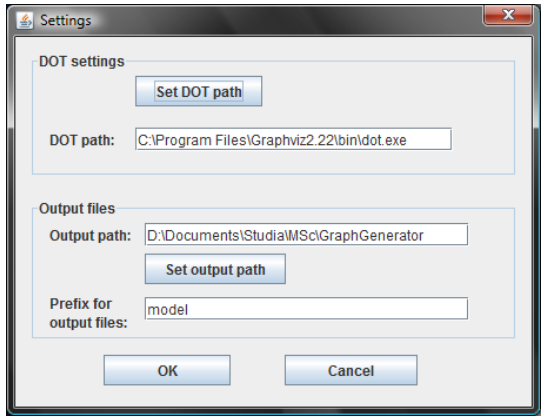


Figure 5.11: Settings window.

Not all properties are set directly in GME. Some program settings may be

changed in *Settings* window, which opens after pressing *Settings* button. The window is presented in figure 5.11.

The field with *DOT path* label specifies location of *dot* program, which is a part of *Graphiz* tool. Output path specifies the directory, where all generated files should be saved. By default it is set to the same directory, where the interpreted meta-model is stored. The last property, which may be set, is a prefix for output files. It is a string that is added in the beginning of each generated model. Full name of generated files is concatenation of prefix, index of generated file (always starting from zero and incrementing with one for each next model) and file extension (`xme`, `dot` or `png`).

Chapter 6

Evaluation of the Implemented Tool

The developed solution has to be verified, in order to prove that it is able to generate the most widely used models, as well as their more complicated modifications. For testing purposes, meta-models of task graphs, Petri net and sequencing graph have been created. The meta-models are used as inputs for the model generator. The generated synthetic models have been checked using GME constraints verifier, as well by reviewing generated *PNG* files.

Section 6.1 contains a description of two task graph meta-models (for systems containing homo- and heterogeneous processors). Section 6.2 presents details of three different Petri net meta-models. Section 6.3 is an overview of sequencing graph for biochips model. The last section presents conclusions and ways of further tool development.

6.1 Task Graph

The meta-model of task graph is the first case study. In order to model it, only one vertex type has to be defined. It is obvious that successor of the type is of the same type. Cycles and self-loops are not allowed. Multiplicity of each graph

edge should be one. Maximum number of edges outgoing from a vertex and incoming to a vertex may be freely defined. In this case, maximum number of incoming edges is set to 3 and there may be 2 or 3 edges incoming to each vertex. To simulate task execution time, a double attribute is associated with the graph vertex type. Its value has a normal distribution, with mean value equal to 2 and variance equal to 1. The graphical representation of the meta-model is presented in picture 6.1.

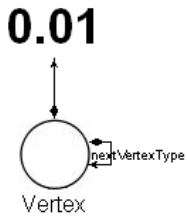


Figure 6.1: Task graph meta-model.

After specification of the meta-model, generation GME meta-model and synthetic models, the meta-model may be registered in GME and models may be loaded into GME environment. A part of the generated model is presented in figure 6.2.

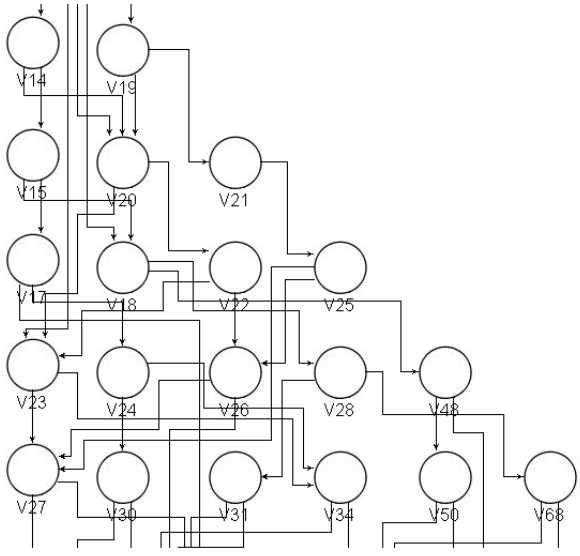


Figure 6.2: A part of a task graph synthetic model.

The synthetic models are associated with automatically generated constraints, which are described in section 5.2.5. The generated models are associated with constraints limiting *ExecTime* attribute values, specifying maximum number of edges incoming to a vertex, as well as minimum and maximum number of edges outgoing from a vertex. The last constraint limits edges multiplicity to 1. All constraints are correct, they always reflect meta-model specification. The generated models are also correct, since they never violate any constraint.

The generated task graph represents a system, which is a network of homogenous processors (execution time has the same characteristic for each node). In order to model system containing heterogeneous processors, two approaches may be used. The first one requires using a different tool, which transforms some task execution times into a different scale. However, the problem may be also solved without using any external tool. Let's assume that the modeled system contains two processor types. The first type is twice as fast as the second type. This situation may be described using two vertex types: *Slower* and *Faster*. In order to show different execution times, *Slower* type is associated with uniformly distributed integer variable, which values belong to range $[2, 10]$. *Faster* type is associated with a different attribute. Its values are also uniformly distributed, however the values belong to range $[1, 5]$. Since tasks may be freely distributed over the system, it is possible that successor of a *Slower* type node is a different *Slower* node, a *Faster* node or a combination of both types. Similarly, successor of *Slower* vertex type may be either a different *Slower* node, a *Faster* type vertex or a couple of *Slower* and a couple of *Faster* nodes. In order to model this situation, four different connection types have to be introduced: from *Slower* to *Slower* type, from *Slower* to *Faster* type, from *Faster* to *Faster* type and from *Faster* to *Slower* type. Since combinations of two different successor types are allowed, `NextVerticesTypes` parameter is set to *XOR* for the both vertex types. The graphical representation of the meta-model is presented in figure 6.3.

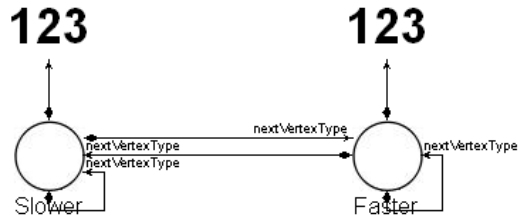


Figure 6.3: Meta-model of a task graph modeling a heterogeneous system.

The generated meta-model is associated with much more constraints, than the

first simple example, since there are four connection types and two vertex types (there are separated constraints for each of them). Apart from the constraints similar to the previous example, a new constraint family is added – constraints checking if there is only one connection type outgoing from each vertex (result of using *XOR* flag). All constraints are generated correctly and they are associated with an appropriate vertex or connection type. Generated models are also correct. They do not violate any constraint and they are a random combination of two different task types. A part of the generated model, loaded into GME environment, is presented in figure 6.4. *Faster* vertex type is represented by grey circles, whereas *Slower* type is represented by white circles.

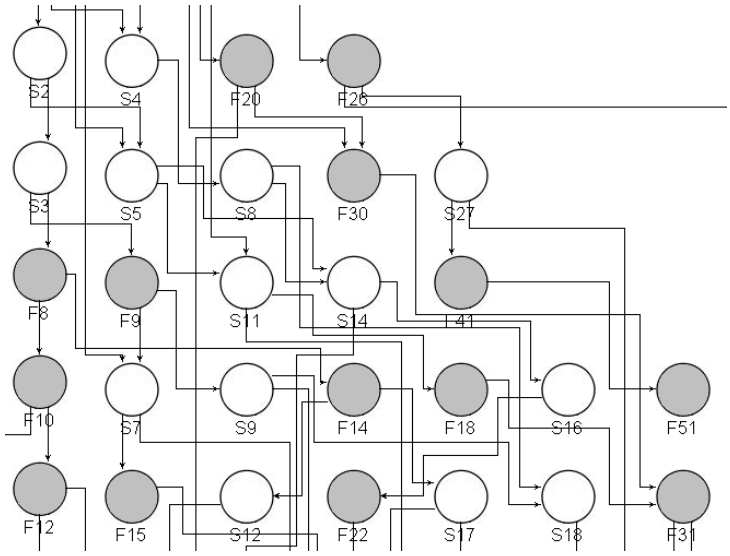


Figure 6.4: A part of a task graph modeling a heterogeneous system.

6.2 Petri Net

The second embedded system model, which we have used for evaluation, is the Petri net model of computation. In order to model it, two vertex types have to be defined: *Place* and *Transition*. The first type may be used by final and starting vertices. *Transition* type can be used neither by starting nor by final vertices. For that reason this case study allows verifying the algorithm, which eliminates non-final type vertices without any outgoing edges.

The vertex types cannot be freely mixed together. Successor of *Place* type

vertex may be only *Transition* type vertex. Each *Transition* type vertex must have *Place* type successor.

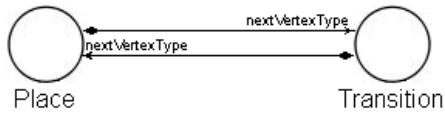


Figure 6.5: Simple Petri net meta-model.

The last property, which is characteristic for each Petri net and has not been tested in the previous examples, is that it is cyclic. In order to generate such a model, `LoopEdge` must be added in *Architecture* aspect. Similarly to task graphs, maximum number of incoming and outgoing edges is system dependent. For testing purposes, maximum number of edges incoming to a vertex (*Transition* or *Place* type) is set to 2 and there are 1 or 2 edges outgoing from each vertex. The meta-model of a Petri net is presented in figure 6.5.

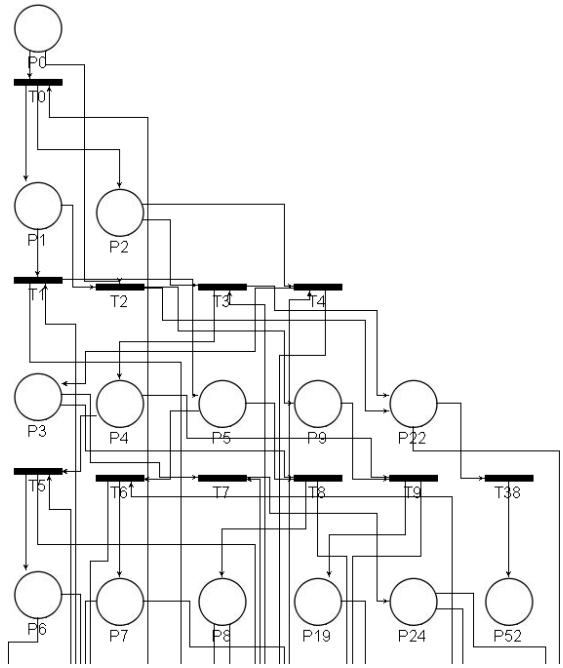
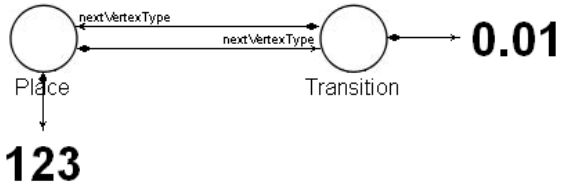


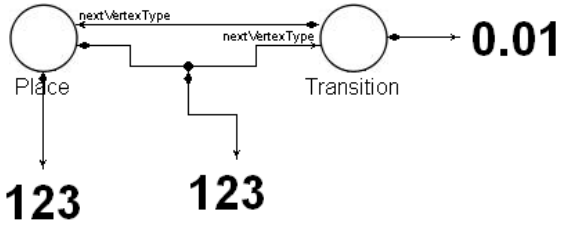
Figure 6.6: A part of a Petri net synthetic model.

The constraints created by the model generator are similar to the previous cases. However, there are two new constraints, which enforce that *Transition*

type vertex can be used neither by starting nor by final vertices. All the generated constraints were correct and associated with appropriate entities. Also the algorithm eliminating not-final type vertices without any outgoing edges works correctly – all the final states are *Place* type. Synthetic models are generated correctly, they do not violate any constraint. Part of a generated model is presented in figure 6.6.



(a)



(b)

Figure 6.7: Meta-models of a marked Petri net with multiple edges represented as: (a) parallel edges, (b) as a parameter associated with a connection type.

The created Petri net meta-model is very simple. As mentioned in section 2.2, Petri net places are marked with tokens and there may be more than one edge connecting a pair of the same transition and place or place and transition. Multiple edges may be presented in two different ways: as multiple, parallel edges outgoing and coming to the same two states or an attribute associated with an edge. In order to test `multiplicity` meta-model parameter and association attributes with connections, two Petri net meta-models are created. In both cases each transition is associated with an execution time and each place is associated with token number. Meta-models of Petri nets with associated attributes are presented in figure 6.7.

Generated models are associated with constraints analogical to the previous Petri net example. For the meta-model, where edge multiplicity is represented as a connection attribute, the constraints on the minimum and maximum values are associated with a connection type (as expected). Also the other constraints

are associated with an appropriate entity. Each synthetic model successfully passed GME validation – models do not violate any constraint. Figure 6.8 presents a part of a synthetic Petri net, where maximum multiplicity edge is set to 2 and multiple edges are represented by parallel arrows. The picture is not a screen shot of model imported to GME environment, but part of *PNG* file generated from *DOT* model representation. The graphical representation of the models, where multiplicity is represented by an integer attribute associated with edges is similar to the figure 6.5. The only difference is that there is an additional attribute, which may be seen in GME, in the property window of a connection.

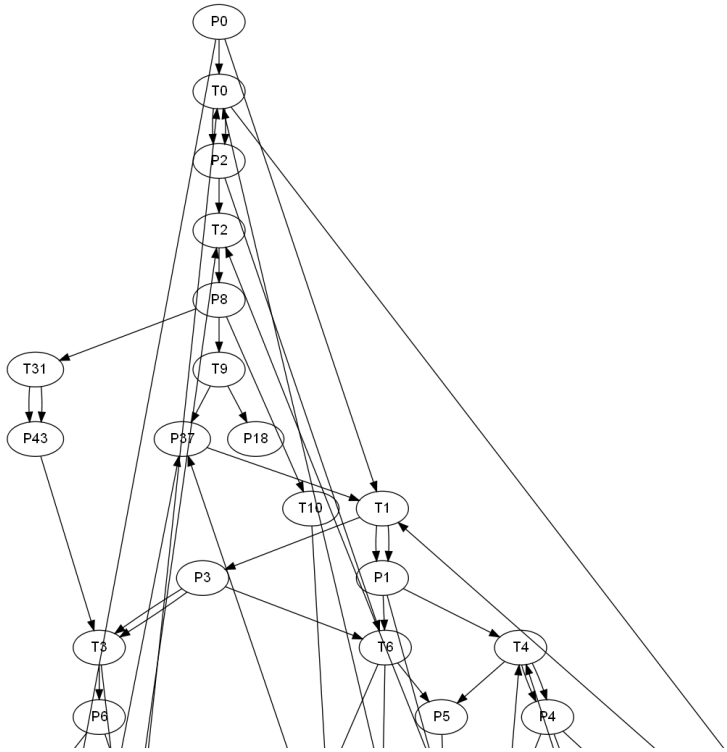


Figure 6.8: A part of a *PNG* representation of Petri net synthetic model.

6.3 Sequencing Graph for Biochips

In order to test the algorithm generating multiple successor or predecessor types, a more advanced model must be used. As mentioned before, sequencing graphs

are models, which may have a lot of rules specifying connection between different node types as well as different requirements of the successor or predecessor types.

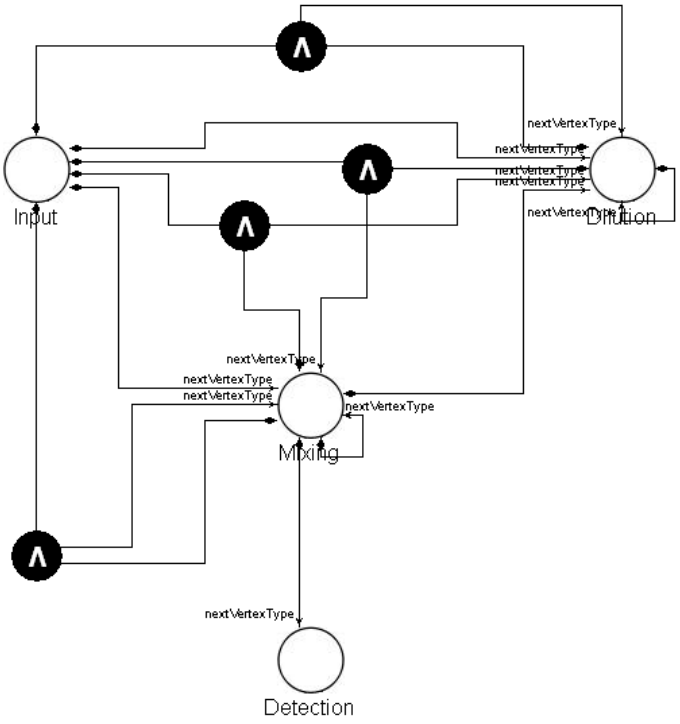


Figure 6.9: Meta-model of a sequencing graph for biochips.

In section 2.4, there is a detailed description of a sequencing graph for biochips. It is composed of four different node types: input, dilution, mixing and detection. Therefore there are four different vertex types defined in the sequencing graph meta-model. Only *Input* type may be used by starting vertices and only *Detection* type nodes may be the final ones. In order to model that *Mixing* vertex type may have two *Mixing* or two *Input* predecessors, two different connection types are defined. They both have `MinOut` and `MaxOut` parameters set to two. In order to present two other predecessor types: a pair of 1 mixing and 1 input or a pair of 1 dilution and 1 input *AND* entity has to be used. All associated `MinOut` and `MaxOut` attributes are set to one.

Dilution type nodes may have one of four predecessor types: two *Mixing* type nodes, two *Input* type nodes, a pair of one *Input* and one *Mixing* type node, as well as a pair of one *Input* and one *Dilution* type node. They are modeled on

the analogical manner, as predecessors of *Mixing* vertex type.

The only type that may be used as a predecessor of *Detection* type, is *Mixing* type. Each *Detection* type node is a successor of a single *Mixing* type node. In order to model it, one more connection type has to be introduced, with `MinOut` and `MaxOut` attributes set to one.

Only one connection type (to a single or a multiple predecessor types) may be used, for each vertex type. In order to achieve that, `NextVerticesTypes` parameter is set to *XOR* for all vertex types. The meta-model of a sequencing graph for biochips is presented in figure 6.9.

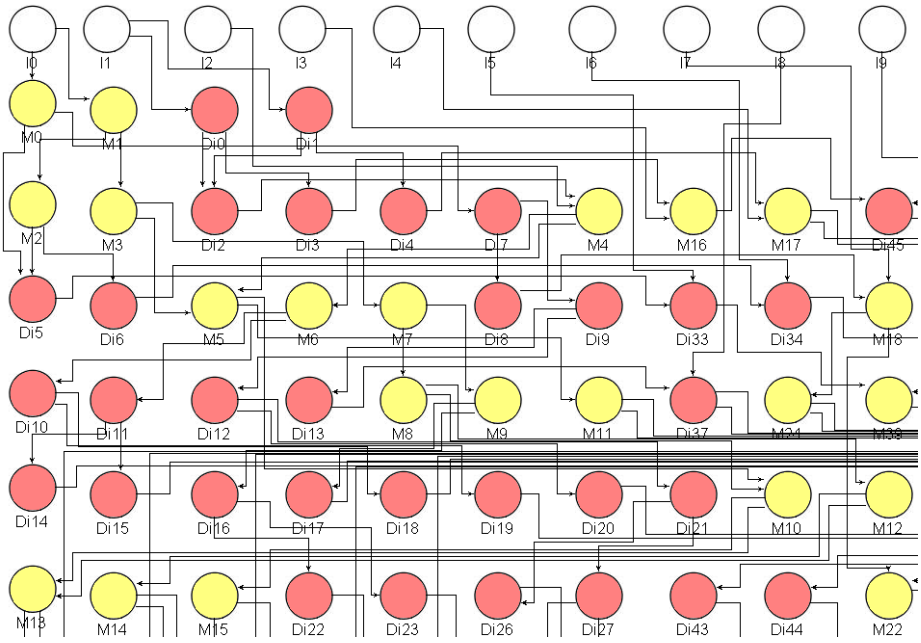


Figure 6.10: A part of a sequencing graph for biochips model.

Generated models are much more complicated than the previous case studies. There are over 60 constraints added to each model. Additionally to the ones tested before (constraints on minimum and maximum number of outgoing edges, edge multiplicity, single outgoing connection type, etc.), constraints enforcing multiple predecessor types are generated. In order to test the constraints, a generated model has been modified – some of the required predecessors were removed. Since validation process failed in such a situation, the constraints are correct. All the constraints are correct and, what is also very important, they are associated with appropriate entities. Synthetic models do not violate any

constraints – they are also generated properly. A part of a generated model, loaded into GME environment, is presented in the figure 6.10.

6.4 Future Work

The developed solution (the generic meta-model for embedded system models and its interpreter) is able to generate advanced, synthetic models of embedded systems. The model generator was successfully verified by the case studies described in the previous sections of this chapter. Although the tool is already quite generic, it may still be extended, in order to increase its usefulness.

The current implementation is able to generate only models with a completely random structure. However, in some situations a more precise structure is required. For example in order to model algorithms based on "divide and conquer" paradigm, a model must have a tree structure [4]. The other algorithms may be modeled better using series-parallel model structure [23].

Generating diverse model structures is not a difficult task. There is a very simple algorithm generating series-parallel graphs using recursion [25]. It starts from two nodes A and B connected with an edge. In each algorithm step, such a graph part is modified. The connection between two graph nodes may be removed and changed by a connection from A to a newly created node X and a connection from X to B (series part is created). The second possibility is adding another connection from A to B node (parallel part is created). Although the algorithm is very easy to implement it is very hard to use it when multiple predecessor or successor types are used. It is also hard to guarantee a minimum number of edges outgoing from a vertex.

Also creating tree structure is a relatively easy task. It may be achieved by building heap structure from the nodes set [7]. However, generating heap structure is possible only when successor number is a constant, not a random value from a given range. It is also very hard to create a heap, when multiple successor or predecessor types are used.

In order to implement different model structures, new entities have to be added to the *Architecture* meta-model aspect. A new, abstract `GraphPart` entity may be introduced. The current `Graph` entity, as well as the entities representing random-parallel and tree parts may inherit from it. User should be able to freely connect the new entities together with each other, as well as connect them to start and final dummy vertex – appropriate connections have to be introduced. Interpreter has to be modified, in order to read and define a model

structure. The structure may be stored by new classes reflecting new meta-model entities. However, by using structure different than the random one, user would miss some functionality, like multiple successor/predecessor types or minimum number of outgoing edges.

Chapter 7

Conclusions

In this thesis we have proposed, designed and implemented a generic model generator, which may be used in many different areas related to embedded system design. The developed solution is generic, but user friendly and quite intuitive. It is achieved by using a completely new approach to the model generators. Current solutions concentrate on one, particular model type. User is able to adjust it with many different kinds of arguments or complicated scripts. Most of the tools do not have a graphical user interface, so command line parameters is the only way of setting-up the program. In order to use a tool, the user has to learn all parameters, their syntax and semantic. If input of a tool is an XML file, the user has to also know the appropriate XML schema, to create a correct input file.

We have proposed entities and attributes for describing a wide range of embedded systems models. Instead of learning many different keywords and parameters, a user can draw a meta-model in a graphical GME environment. The user may define multiple vertex types, connection rules and attributes, associated with entities. Each vertex type may be associated with multiple number of successor or predecessor vertex types. Defining a new node type or a connection rule is straightforward: the user just drags a suitable entity from the pallet.

After defining a meta-model, synthetic models are created by the Model Generator. We have proposed and implemented an algorithm for synthetic model

generation, which takes as the input the embedded system meta-model specified by the user. A model structure may be very simple (with single vertex and connection type), as well as very complex, with many vertex kinds, multiple predecessor or successor types, etc. Each vertex and connection type may be associated with any number of attributes. Each attribute may be produced by a random generator with uniform, exponential or normal value distribution.

The Model Generator creates not only synthetic models, but also a GME representation of the meta-model, defined by the user. It can be used to customize GME such that the generated models can be modified or a new model may be created manually, in a graphical environment.

modify the synthetic models or to create a new model manually in user-friendly GME environment.

As mentioned in the previous chapter, there are still some tool features that may be implemented or extended. However, we believe that the tool in the current development state may be useful and it will help at least researchers belonging to Embedded Systems Engineering (ESE) section at Informatics and Mathematical Modeling department of DTU. The tool has been evaluated on several embedded system models and has been successfully used by the researchers in ESE to generate synthetic sequencing graphs for biochemical applications.

Bibliography

[1] Metaedit+ homepage. `http://www.metacase.com` [cited 7 May 2009].

[2] The eclipse modeling framework (emf) overview. `http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html` [cited 7 May 2009], June 2005.

[3] Alexandro M. S. Adário and Sergio Bampi. Extending sequencing graphs for reconfigurable applications modeling. In *SBCCI '01: Proceedings of the 14th symposium on Integrated circuits and systems design*, pages 161–166, Washington, DC, USA, 2001. IEEE Computer Society.

[4] V. A. F. Almeida, I. M. M. Vasconcelos, J. N. C. Árabe, and D. A. Menascé. Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 683–691, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[5] Said Amellal and Bozena Kaminska. Scheduling of a control and data flow graph. *Circuits and Systems, ISCAS '93, 1993 IEEE International Symposium on*, 3:1666–1669, 1993.

[6] Philippe Chrétienne, Edward G. Coffman, Jan Karel Lenstra, and Zhen Liu, editors. *Scheduling Theory and Its Applications*. Wiley, New York, 1995.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[8] Robert P. Dick, David L. Rhodes, and Wolf Wayne. Tgff: task graphs for free. In *Hardware/Software Codesign, 1998. (CODES/CASHE apos;98) Proceedings of the Sixth International Workshop on*, pages 97–101, 1998.

[9] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4:686–701, 1990.

[10] Institute for Software Integrated Systems, Vanderbilt University. *High-Level Java Interface to GME*, 1.0 edition, 2004.

[11] Institute for Software Integrated Systems, Vanderbilt University. *GME 7 User's Manual*, 7.0 edition, 2007.

[12] Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[13] E. A. De Kock, G. Essink, W. J. M. Smits, and P. Van Der Wolf. Yapi: Application modeling for signal processing systems. In *In Proc. 37th Design Automation Conference (DAC'2000*, pages 402–405. ACM Press, 2000.

[14] Yu kwong Kwok, Ishfaq Ahmad, and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7:506–521, 1996.

[15] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. *The Generic Modeling Environment.* Nashville, TN 37235, USA, 2001.

[16] Insup Lee, Joseph Y-T. Leung, and Sang Son, editors. *Handbook of Real-Time and Embedded Systems.* CRC Press, Boca Raton, FL, USA, 2007.

[17] Peter Marwedel. *Embedded system design.* Springer, 2006.

[18] C.L. McCreary, A. A. Khan, J. Thompson, and M.E. McArdle. A comparison of heuristics for scheduling dags on multiprocessors. In *in Proceedings of the Eighth International Parallel Processing Symposium*, pages 446–451, 1994.

[19] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580, 1989.

[20] ARTEMIS Office. Strategic research agenda. Technical report, ARTEMIS Programme, 2006.

[21] Jagdish K. Patel and Campbell B. Read. *Handbook of the normal distribution*. Mercel Dekker, inc., 1996.

[22] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. John Wiley and Sons, 1999.

[23] Robin A. Sahner and Kishor S. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Trans. Softw. Eng.*, 13(10):1105–1114, 1987.

[24] V. Sarkar. *Partitioning and scheduling parallel programs for execution on multiprocessors*. PhD thesis, Stanford, CA, USA, 1987.

[25] Berry Schoenmakers. A new algorithm for the recognition of series-parallel graph. Technical report, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1995.

[26] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using kahn process networks: The compaan/laura approach. In *In Proceedings of the Design, Automation and Test in Europe Conference*, pages 1–6, 2004.

[27] Sander Stuijk, Marc Geilen, and Twan Basten. Sdf3: Sdf for free. In *ACSD '06: Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 276–278, Washington, DC, USA, 2006. IEEE Computer Society.

[28] K. Thulasiraman and M.N.S. Swamy. *Graphs: Theory and Algorithms*. Wiley-Interscience, April 1992.

[29] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multi-processor scheduling algorithms. *Journal of Scheduling*, (5):379–394, 2002.

[30] Keith Vallerio. *Task Graphs for Free (TGFF v3.0)*, April 2008.

[31] Keith S. Vallerio and Niraj K. Jha. Task graph extraction for embedded system synthesis. In *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*, pages 480–486, Washington, DC, USA, 2003. IEEE Computer Society.

[32] Tao Xu and Krishnendu Chakrabarty. Automated design of digital microfluidic lab-on-chip under pin-count constraints. In *ISPD '08: Proceedings of the 2008 international symposium on Physical design*, pages 190–198, New York, NY, USA, 2008. ACM.

[33] Tao Xu and Krishnendu Chakrabarty. Broadcast electrode-addressing for pin-constrained multi-functional digital microfluidic biochips. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 173–178, New York, NY, USA, 2008. ACM.

[34] Tao Yang and Apostolos Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:951–967, 1994.

[35] Richard Zurawski, editor. *Embedded Systems Handbook*. Taylor and Francis Group, 2006.

[36] Richard Zurawski and MengChu Zhou. Petri nets and industrial applications: A tutorial. *Industrial Electronics, IEEE Transactions on*, 41:567–583, 1994.

Appendix A

GME meta-model and GME model XML Schemas

Figure A.1: XML Schema of a GME meta-model in XML format

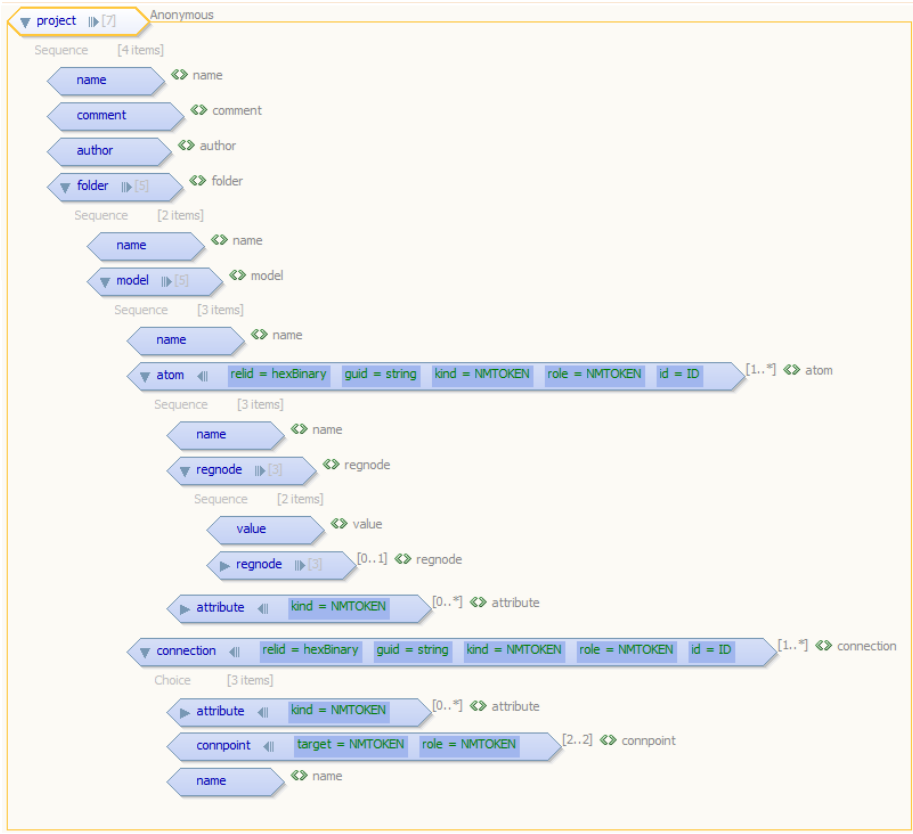
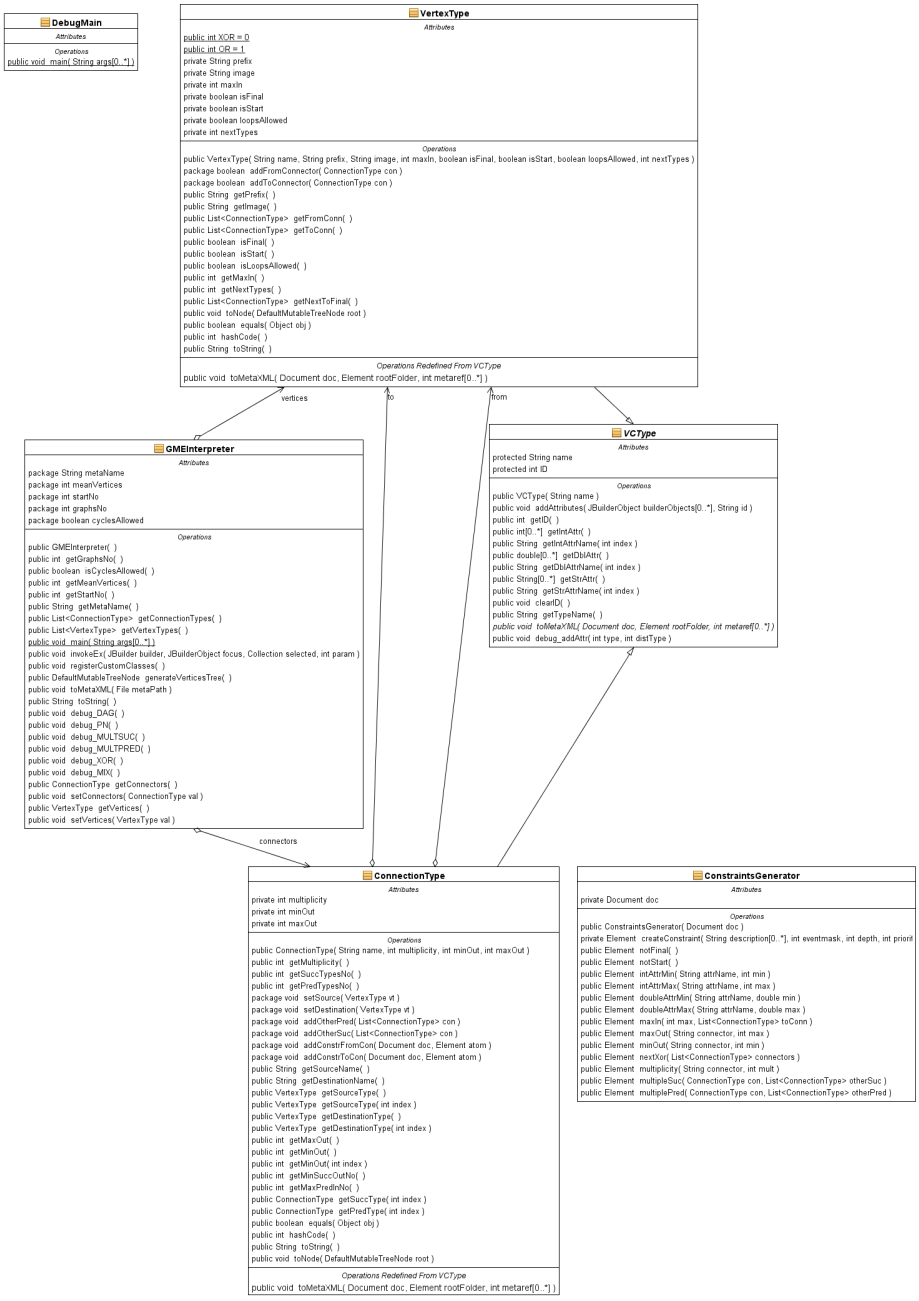


Figure A.2: XML Schema of a GME model in XML format

Appendix B

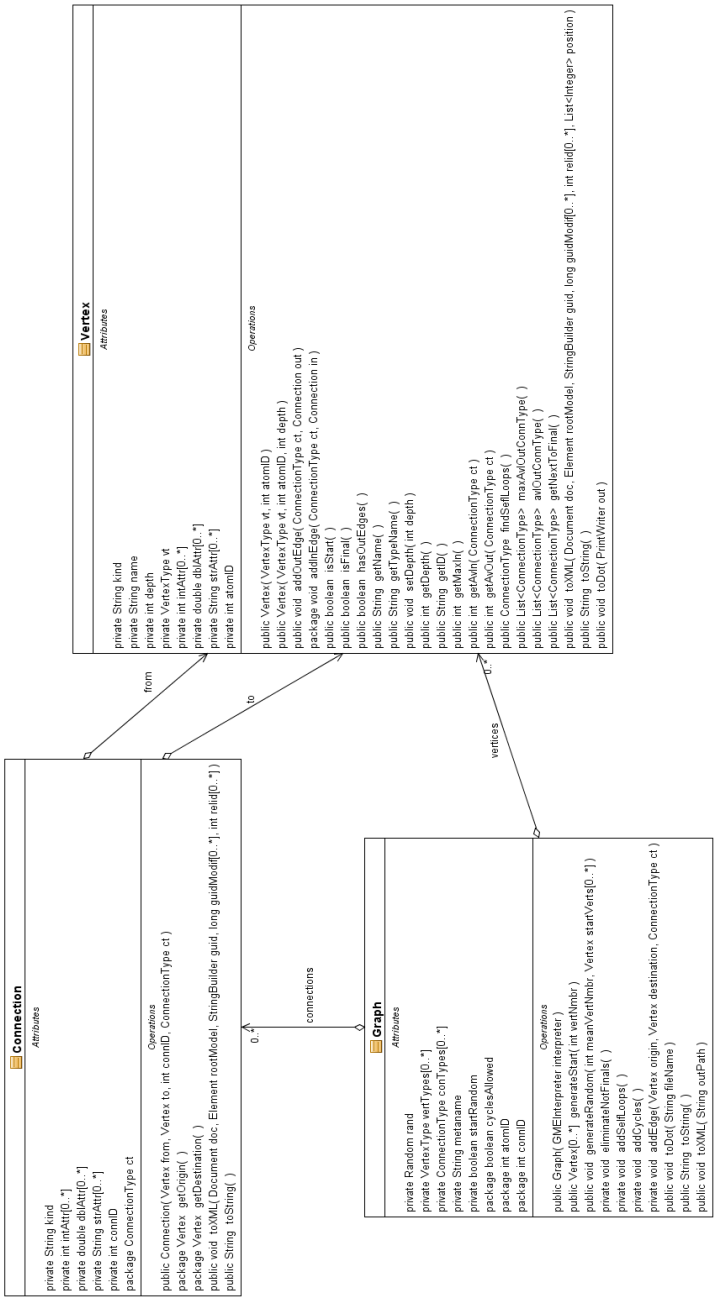
Class Diagram of the GME Plug-in

B.1 `Interpreter` package

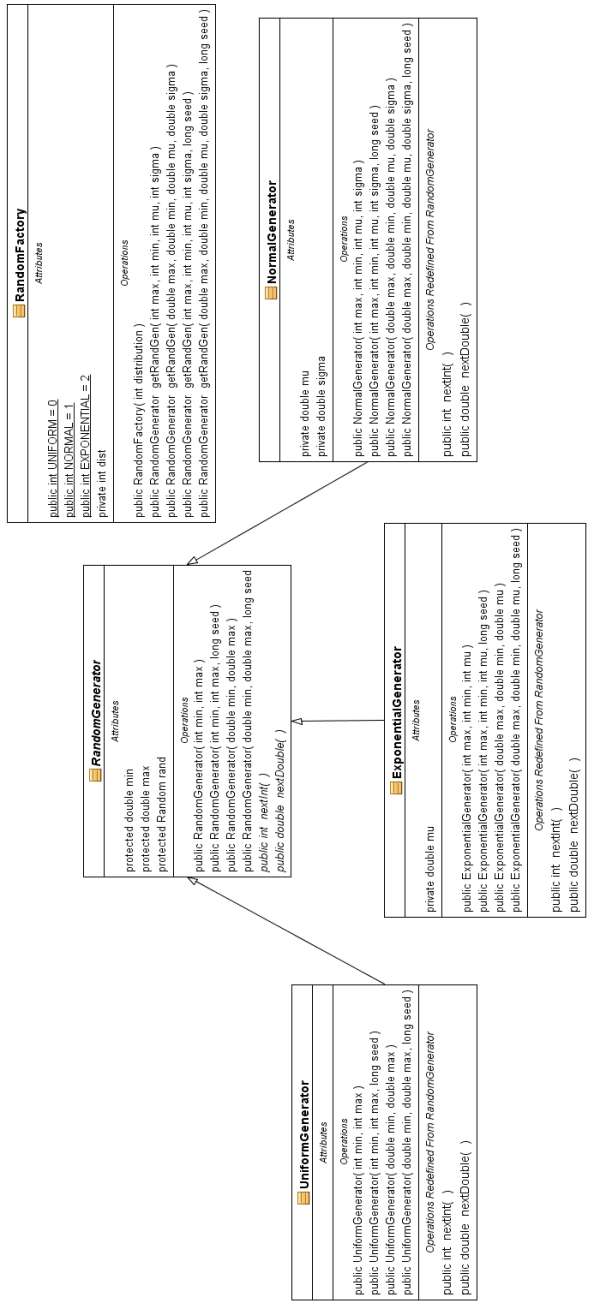

DebugMain
Attributes
Operations
public void main(String args[0..*])
VertexType
Attributes
public int XOR = 0
public int OR = 1
private String prefix
private String image
private int maxIn
private boolean isFinal
private boolean isStart
private boolean loopsAllowed
private int nextTypes
Operations
public VertexType(String name, String prefix, String image, int maxIn, boolean isFinal, boolean isStart, boolean loopsAllowed, int nextTypes)
package boolean addFromConnector(ConnectionType con)
package boolean addToConnector(ConnectionType con)
public String getPrefix( )
public String getImage( )
public List<ConnectionType> getFromConn( )
public List<ConnectionType> getToConn( )
public boolean isFinal( )
public boolean isStart( )
public boolean isLoopsAllowed( )
public int getMaxIn( )
public int getNextTypes( )
public List<ConnectionType> getNextToFinal( )
public void toNode(DefaultMutableTreeNode root)
public boolean equals(Object obj)
public int hashCode( )
public String toString( )
Operations Redefined From VCType
public void toMetaXML(Document doc, Element rootFolder, int metaref[0..*])
vertices
to
from
GMEInterpreter
Attributes
package String metaName
package int meanVertices
package int startNo
package int graphsNo
package boolean cyclesAllowed
Operations
public GMEInterpreter( )
public int getGraphsNo( )
public boolean isCyclesAllowed( )
public int getMeanVertices( )
public int getStartNo( )
public String getMetaName( )
public List<ConnectionType> getConnectionTypes( )
public List<VertexType> getVertexTypes( )
public void main(String args[0..*])
public void invokeEx(JBuilder builder, JBuilderObject focus, Collection selected, int param)
public void registerCustomClasses( )
public DefaultMutableTreeNode generateVerticesTree( )
public void toMetaXML(File metaPath)
public String toString( )
public void debug_DAG( )
public void debug_PN( )
public void debug_MULTSUC( )
public void debug_MULTPRED( )
public void debug_XOR( )
public void debug_MIX( )
public ConnectionType getConnectors( )
public void setConnectors(ConnectionType val)
public VertexType getVertices( )
public void setVertices(VertexType val)
VCType
Attributes
protected String name
protected int ID
Operations
public VCType(String name)
public void addAttributes(JBuilderObject builderObjects[0..*], String id)
public int getID( )
public int[0..*] getIntAttr( )
public String getIntAttrName(int index)
public double[0..*] getDblAttr( )
public String getDblAttrName(int index)
public String[0..*] getStrAttr( )
public String getStrAttrName(int index)
public void clearID( )
public String getTypeName( )
public void toMetaXML(Document doc, Element rootFolder, int metaref[0..*])
public void debug_addAttr(int type, int distType)
connectors
ConnectionType
Attributes
private int multiplicity
private int minOut
private int maxOut
Operations
public ConnectionType(String name, int multiplicity, int minOut, int maxOut)
public int getMultiplicity( )
public int getSuccTypesNo( )
public int getPredTypesNo( )
package void setSource(VertexType vt)
package void setDestination(VertexType vt)
package void addOtherPred(List<ConnectionType> con)
package void addOtherSuc(List<ConnectionType> con)
package void addConstrFromCon(Document doc, Element atom)
package void addConstrToCon(Document doc, Element atom)
public String getSourceName( )
public String getDestinationName( )
public VertexType getSourceType( )
public VertexType getSourceType(int index)
public VertexType getDestinationType( )
public VertexType getDestinationType(int index)
public int getMaxOut( )
public int getMinOut( )
public int getMinOut(int index)
public int getMinSuccOutNo( )
public int getMaxPredInNo( )
public ConnectionType getSuccType(int index)
public ConnectionType getPredType(int index)
public boolean equals(Object obj)
public int hashCode( )
public String toString( )
public void toNode(DefaultMutableTreeNode root)
Operations Redefined From VCType
public void toMetaXML(Document doc, Element rootFolder, int metaref[0..*])
ConstraintsGenerator
Attributes
private Document doc
Operations
public ConstraintsGenerator(Document doc)
private Element createConstraint(String description[0..*], int eventmask, int depth, int priorit
public Element notFinal( )
public Element notStart( )
public Element intAttrMin(String attrName, int min)
public Element intAttrMax(String attrName, int max)
public Element doubleAttrMin(String attrName, double min)
public Element doubleAttrMax(String attrName, double max)
public Element maxIn(int max, List<ConnectionType> toConn)
public Element maxOut(String connector, int max)
public Element minOut(String connector, int min)
public Element nextXor(List<ConnectionType> connectors)
public Element multiplicity(String connector, int mult)
public Element multipleSuc(ConnectionType con, List<ConnectionType> otherSuc)
public Element multiplePred(ConnectionType con, List<ConnectionType> otherPred)

Attribute
Attributes
public int UNIFORM = 0
public int NORMAL = 1
public int EXPOTENTIAL = 2
protected String name
Operations
public Attribute(String name)
public String getName( )
public void toMetaXML(Document doc, Element rootFolder, Element vcType, int metaref[0..
public void toNode(DefaultMutableTreeNode root)
StringAttribute
Attributes
private String value
Operations
public StringAttribute(String name, String value)
public String getValue( )
public String toString( )
Operations Redefined From Attribute
public void toMetaXML(Document doc, Element rootFolder, Element vcType, int met
public void toNode(DefaultMutableTreeNode root)
DoubleAttribute
Attributes
private int distribution
private double max
private double min
private double mean
private double variance
private RandomGenerator gen
Operations
public DoubleAttribute(String name, int distribution, double max, double min, double mean, double varian
public double getValue( )
public String toString( )
Operations Redefined From Attribute
public void toMetaXML(Document doc, Element rootFolder, Element vcType, int metaref[0..*])
public void toNode(DefaultMutableTreeNode root)
IntAttribute
Attributes
private int distribution
private int max
private int min
private int mean
private int variance
private RandomGenerator gen
Operations
public IntAttribute(String name, int distribution, int max, int min, int mean, int variance)
public int getValue( )
public String toString( )
Operations Redefined From Attribute
public void toMetaXML(Document doc, Element rootFolder, Element vcType, int meta
public void toNode(DefaultMutableTreeNode root)

B.2 `Generator` package



B.3 Random package



B.4 GUI package