Final Thesis

Towards an Extensible Visual Editor for Embedded Systems Representation Models

by

Jirong Zhu

LITH.IDA-Ex--03/061--SE

2003-10-23

ABSTRACT

Nowadays, embedded systems are widely used everywhere, and their complexity is constantly growing. A very important aspect in embedded systems design is their modeling, which describes the system functionality and structure at several levels of abstraction and stages of design.

There are many types of representation models used for the modeling of embedded systems, such as flow charts, entity relationship diagrams, call trees, etc. This thesis is concerned with process graphs, statecharts and petri nets, the representation models that are commonly utilized in embedded systems design.

Research has shown that productivity is enhanced by the use of domain-specific tools that allow the visual manipulation of models. Instead of working with a text-based representation of the model, using such tools, the users can design and edit the models visually. However, the development of such tools is very time-consuming. Moreover, new models are continuously introduced by the industry and the research community. Many are completely new, but most are the variations of existing models.

In order to speed up the development of domain-specific visual model editors, generic-modeling environments, which can be extended, have to be developed. In this thesis we propose a generic modeling environment, called Extensible Visual Editor (EVE), which can be extended to handle the class of graph-based embedded systems representation models.

The thesis surveys several domain-specific visual editors for embedded systems representation models. We compare two graph libraries, the Graph Editing Framework (GEF) and the JGraph library. The feasibility of a generic graph-based modeling environment is accessed by implementing a domain-specific editor for conditional process graphs, the CPGEditor. Using experience gained in the design and implementation of the CPGEditor, we give an approach to the design of EVE, including its software structure and functionality. We show how EVE can be extended by the use of model specification files, and we develop specification files for the conditional process graph, the UML statecharts and petri nets.

Acknowledgement

I am thankful to Petru Eles and all my colleagues at Embedded Systems Lab for providing me a pleasant working environment. My special thanks to Paul Pop whose keen supervision and invaluable guidance at every stage helped me a lot in carrying out this thesis work.

Contents

1 Introduction

1.1 Overview

An embedded system is a specialized computer system, which is a part of another system, called the host system. It is designed for a particular kind of application, and its functionality is fixed, it cannot be extended by an end-user through programming.

Embedded systems are now everywhere, from microwave ovens to aircraft-control systems. Because they can be used in such a wide range of products, embedded system may need to meet widely divergent criteria [24]:

- Complex functionality: embedded systems often have to run complicated algorithms and provide complicated user interface.

- Real-time operation: embedded systems often have hard or soft deadlines in completing the operations. A system can fail to meet a soft deadline sometimes, leading to performance degradation. Hard deadlines must be fulfilled always. If any hard deadline is missed, this can lead to catastrophic situations.

- Low manufacturing cost: embedded systems must have low manufacturing costs to increase the market usage.

- Low power: as embedded systems become more autonomous and mobile, battery life becomes very important, which leads to the need of low power architectures.

Nowadays, embedded systems are been widely used everywhere, and their complexity is constantly growing. A very important aspect in embedded systems design is their modeling, which describes the system functionality and structure at several levels of abstraction and stages of design.

There are many types of representations used for the modeling of embedded systems, such as flow charts, entity relationship diagrams, call trees, process graphs, statecharts, petri nets, etc. An important part of such representation models deal with the behavioral description a system, which specifies what functions a system should implement.

Among the representation models, graph-based representations are the most

popular in embedded system design. This thesis is concerned with conditional process graphs (CPG), UML statecharts and petri nets, the representation models that are commonly utilized in embedded systems design.

Research has shown that productivity is enhanced by the use of domain-specific tools that allow the visual manipulation of models [4]. Instead of working with a text-based representation of the model, using such tools, the users can design and edit the models visually. However, the development of such tools is very time consuming. Moreover, the industry and the research community continuously introduce new models. Many are completely new, but most are variations of existing models.

In order to speed up the development of domain-specific visual model editors, generic-modeling environments, which can be extended, have to be developed. In this thesis we propose a generic modeling environment, called Extensible Visual Editor (EVE), which can be extended to handle the class of graph-based embedded systems representation models.

Several editors for different types of graphs have already been developed. For example, Predator editor [19] and PIPE [18] are designed for petri nets. But if the users need to edit variant types of the graphs at one time, they have to load other visual editors with possibly completely different features. So it is very practical and facilitating to build the graph editor into an extensible one, ‘cluster’ the similar graphs in embedded systems and operate on them in one extensible editor.

The thesis first introduces the background knowledge about the embedded systems design and important representation models. We present a brief survey of currently popular graph edit tools and libraries, and provide the analysis of their features.

We compare two graph libraries, the Graph Editing Framework (GEF) and the JGraph library. The feasibility of a generic graph-based modeling environment is accessed by implementing a domain-specific editor for conditional process graphs, the CPGEditor. Using experience gained in the design and implementation of the CPGEditor, we give an approach to the design of EVE, including its software structure and functionality. We show how EVE can be extended by the use of model specification files, and we develop specification files for conditional process graph, UML statecharts and petri nets.

1.2 Contributions

The goal of this thesis is to design a generic modeling environment for graph-based representation models, called Extensible Visual Editor (EVE). EVE is extensible by the use of model specification files. By reading these specifications, the EVE understands how to present the models graphically, and then particularize itself for different representation models, such as CPGs, FSMs (Finite State Machine), petri nets and other types of models.

The contributions of the thesis are:

- Brief survey of the important representation models. There are a variety of models being developed and used to represent the embedded systems, in this thesis we select three typical models: CPG, UML statechart and petri net and briefly describe them given an example for each.

- Survey of related work on graph/diagram editors. According to a variety of representing models there are respectively a variety of editors. In this thesis we mainly introduce two commonly utilized editors that have relatively complete function, one is Microsoft Visio, the other one is Dia.

- Evaluate two graph libraries, GEF and JGraph, including their main features, and functions, their advantages and disadvantages.

- Implementation of a prototype editor for conditional process graphs based on JGraph library.

- The experience gained from the prototype implementation is used to propose the EVE design. EVE is extended using representation-model specification files. We give concrete examples of such files for several representation models.

2 Embedded Systems Design

An embedded computer system uses software running on a hardware platform to implement the system functions. Creating an embedded system, which meets its performance, cost, and design time goals, is a hardware-software co-design problem, that is, the design of the hardware and software components influence each other.

Embedded systems need to fulfill widely different criteria. We have different important design requirements: time constraints, manufacturing cost, modifiability, reliability and so on.

2.1 Tasks of Embedded Systems Design

Embedded system design can be divided into four major tasks:

- Specifying the system functionality using representation models.
- Partitioning the function to be implemented into smaller, interacting pieces.
- Allocating those partitions to microprocessors or other hardware units, where the function may be implemented directly in hardware or in software running on a microprocessor.
- Scheduling the times at which functions are executed, which is important when several functional partitions share one hardware unit [22].

2.2 Design Flow

As shown in Figure 1 (presented in [25]), the design consists of two flows: one is the mapping of the computational parts of the specification onto processing elements of system architecture, and the other is the mapping of the communication in the specification onto system busses. Each flow requires allocation of components, partitioning of the specification onto components, and scheduling of execution on the inherently sequential components. The result is the system architecture of process components connected via busses. Then each component is further implemented through software and hardware synthesis [1].

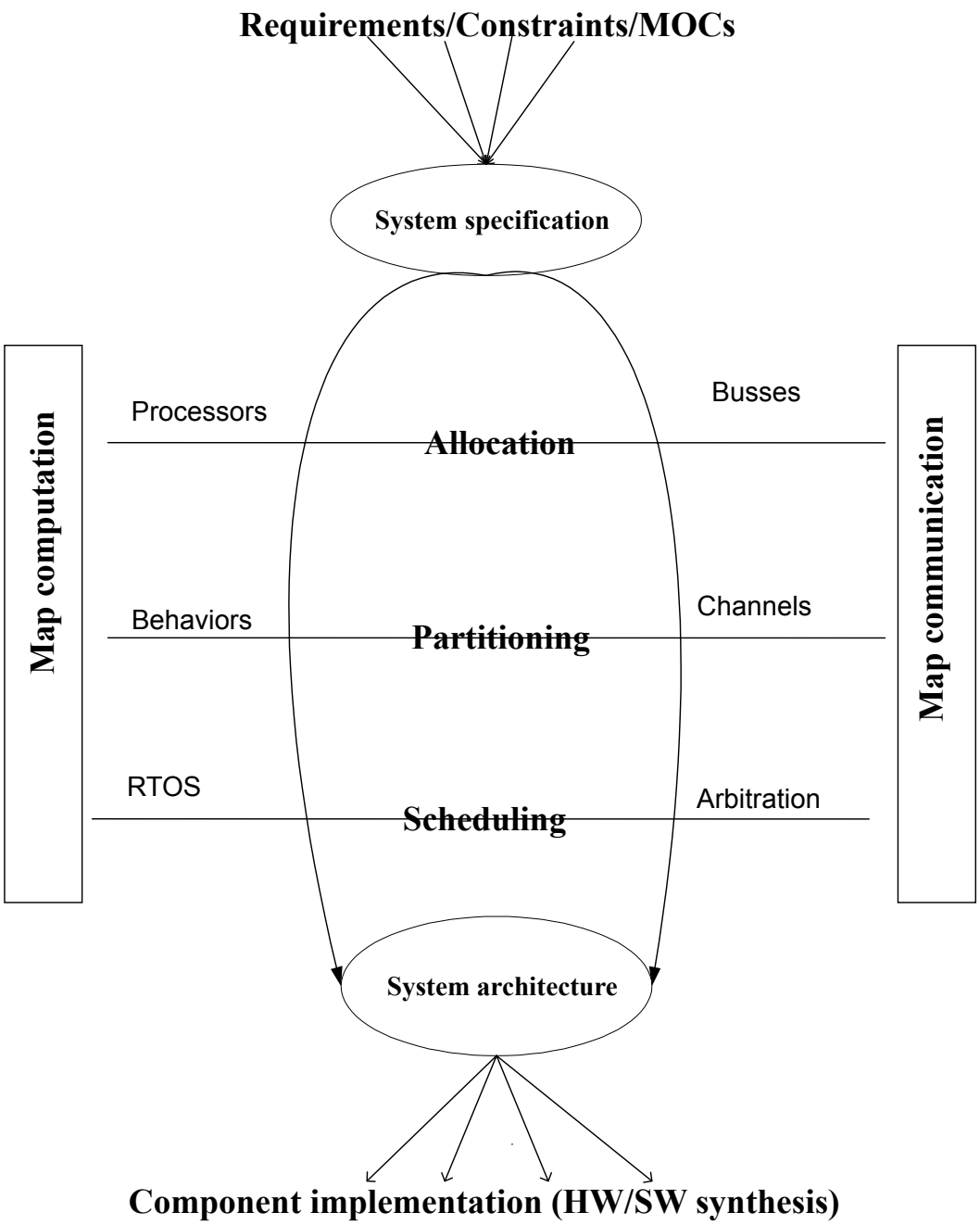


Figure 1.Embedded system design tasks

3 Representation Models

3.1 Introduction

In methodical design of embedded systems, one key aspect is the creation of the models. The models are the concrete representation of knowledge and ideas about a system being developed, and help deal with the complexity of the system.

The purpose of a representation model is to provide a view of the system, and capture the features of a system, describe its functionality. The good model qualities are: it should be formal, no ambiguity (clearly defined syntax and semantics), completed, comprehensible, executable, and easy to use, etc.

The popular models for embedded systems are FSM, dataflow graphs, petri nets, synchronous models, discrete-event systems and so on [12]. In this thesis we mainly focus on CPG, UML statechart and petri net, while the first two are important models belonging to dataflow graphs and FSM respectively.

3.2 Representation Models Based on Graph Structures

3.2.1 Background of Graph Theory

A graph is a diagram composed of a set of points connected by line segments, points are usually called vertices (V) or nodes, and line segments are called edges (E). A vertex (node) is a terminal point or an intersection point of a graph. It is the abstraction of a location. An edge e is a link between two nodes. A link is the abstraction of a transport infrastructure supporting movements between nodes. The link can also be directed or undirected, commonly we represent the directed link with an arrow.

If in a graph there is only one edge joining a pair of vertices and a vertex cannot be adjacent to itself, this kind of graph is called “simple graph”. A simple graph has no arrows, no loops, and cannot have multiple edges joining vertices. In other cases, if there are multiple or parallel edges, then it is “multiple graph”. Depending on different criteria, we have different graph categories [20].

3.2.2 Example of a General Graph

We use an example as the network consisting of the major tourist cities in China (see Figure 2), we just choose six famous ones from them which are BeiJing, XiAn, ShangHai, ChongQing, GuiLin, XiaMen. They can be considered as six nodes in the graph. To facilitate the tourists, we connect these cities for each by line segments in order to show the distances by air.

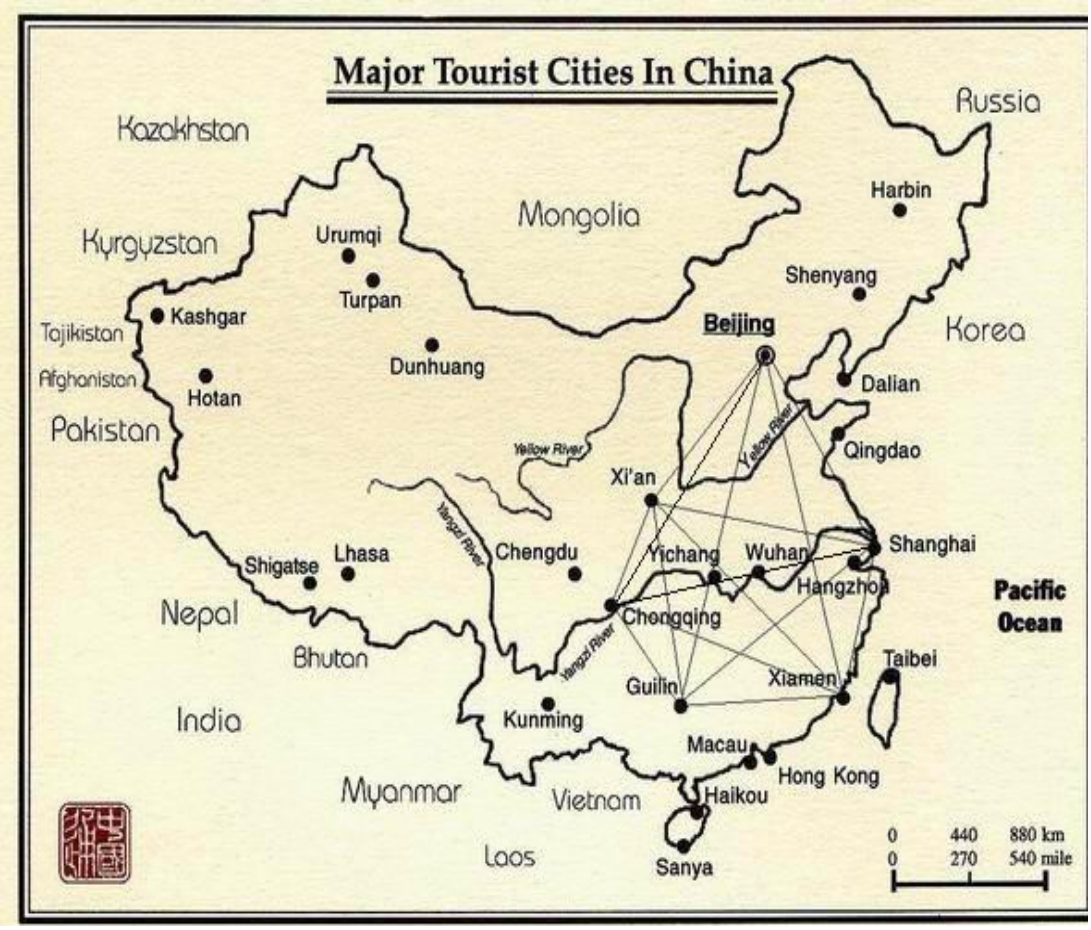


Figure 2. Map of the major tourist cities in China

The graph is represented in Figure 3.

The numbers on the lines are the direct distances between the six cities here, and the unit is km.

In this example, the graph we created is a very simple one, and each pair of nodes in the graph are connected by only one edge, so it is not a multiple graph and not a directed graph either because the airline is not single directed.

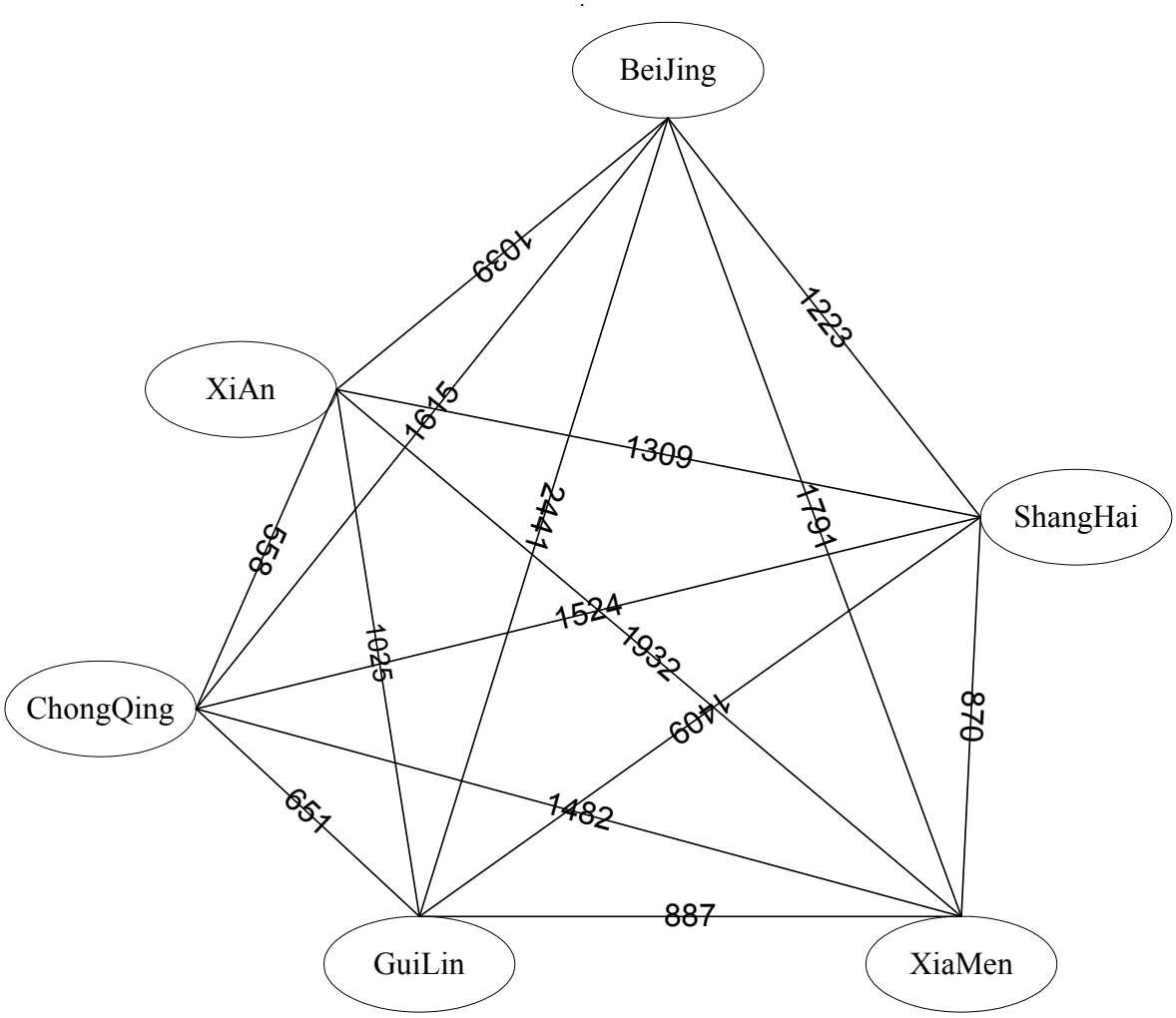


Figure 3. Example graph of the network constitutes of six main tourist cities in China

3.3 Finite State Machine (FSM) and UML Statechart

3.3.1 Background of FSM and UML Statechart

The classical FSM is a well-known model for describing embedded systems. FSM represents a very powerful way of describing and implementing the control logic for systems.

A Finite State Machine is an abstract machine that defined a finite set of condition existence called states, a set of behaviors or actions performed in each of those states, and a set of events that cause changes in states. Each

state of a FSM has transitions to zero or more states. Computation begins in the start state with an input string. It changes to new states depending on the transition functions. The value of the input decides what is the next state with another behavior. The FSM can be considered as a type of directed graph.

One of FSM's disadvantages is that the number of the states is exponentially growing as the system complexity rises. And FSM does not allow the states to be in a hierarchical structure [12].

Unified Modeling Language (UML) is very reliable to support FSM in documenting both different states that a class goes through in embedded systems, and the events that cause changes among those states.

Generally a UML statechart has four elements, and their main functions respectively are: The state marks a mode of the entity. The transition marks the changing of the object state, caused by an event. The initial state is a state of an object before any transitions and only one initial state is allowed on a UML statechart. The final state marks the destruction of the objects whose state we are modeling [3]. Later in the summary we could see their respective represented figures.

3.3.2 Example of a UML Statechart

Here we give an example UML statechart in Figure 4 that models the status of a user's account in a Bug Tracker system [27]:

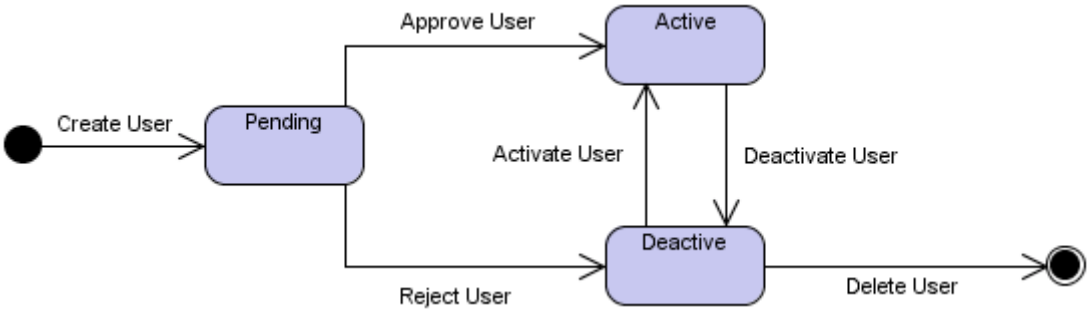


Figure 4. UML statechart example

In the example, from the initial state we create the users and change the state into pending. Then there are two transitions being created, and two events--approve user and reject user--cause the changes, thus it goes to two new states, active and deactive. Between these two states there are mutual functions to change the modes. The final state destructs the deactive user after the users are deleted.

3.4 Conditional Process Graph (CPG)

3.4.1 Background of Process Graph and CPG

A process graph is an abstract representation consisting of a directed, acyclic, and polar graph. Each node is a process, which is a sequence of instructions with worst-case execution time. The processes are either assigned to programmable processors or assigned to hardware processors (ASIC-Application Specific Integrated Circuit). An edge between a pair of nodes indicates the output of one process and the input of the other process, and the edges are communication channels assigned to buses.

Based on the concept of the process graph, a conditional process graph (CPG) is defined in [15]. The graph is conditional because it has the conditional relationship between some processes with an associated condition. Transmission on the edge takes place only if the associated condition is satisfied.

3.4.2 CPG Nodes

- Source and sink nodes

The source and sink in a CPG are dummy nodes (not allocated to any processor, with zero execution time) representing the first and the last processes. All the other nodes in CPG are successors of the source and predecessors of the sink respectively.

- Communication nodes

The communication nodes are introduced for each connection which links processes mapped to different processors. The nodes represented with black dots in Figure 5 are communication processes. The communication processes show inter-processor communication with the execution time, depicted on their left, and the execution time is equal to the corresponding communication time.

- Disjunction nodes

Besides the source, sink and the communication nodes, most nodes are "ordinary nodes" as in Figure 5 represented by solid circles. But among them, a node with conditional edges at its output is a disjunction node, for example, the node between strings "Speed Up" and "Speed Down" in Figure 5, and the corresponding process is called disjunction process.

- Conjunction nodes

A disjunction process has one associated condition, and the complementary values of the condition it computes are on the alternative paths starting from the disjunction node, these paths are disjoint and they meet in a conjunction node with a corresponding process being called a conjunction process. For example, form the bottom up the second node is a conjunction one. A conjunction process can be activated after messages coming on one of the alternative paths have arrived while a non-conjunction process can only be activated after all its inputs have arrived [15].

3.4.3 CPG Edges

- Normal edge

Normal edges are edges without associated conditions. Most edges in Figure 5 are normal edges.

- Conditional edge

An edge is a conditional edge (thick lines in Figure 5) if it has an associated condition. Only when the associated condition is satisfied, the edge would take the transmission between the processes. Conditional edges are always the outputs of the disjunction nodes [16].

3.4.4 CPG Example

We present a CPG example in the area of automotive electronics. The automotive electronics area deals with the electronically controlled functions onboard vehicles.

On long car journeys drivers find it very tiring to keep up continuous pressure on the accelerator pedal. To avoid it many cars now have a system called Cruise Controller (CC). It is a widely used embedded

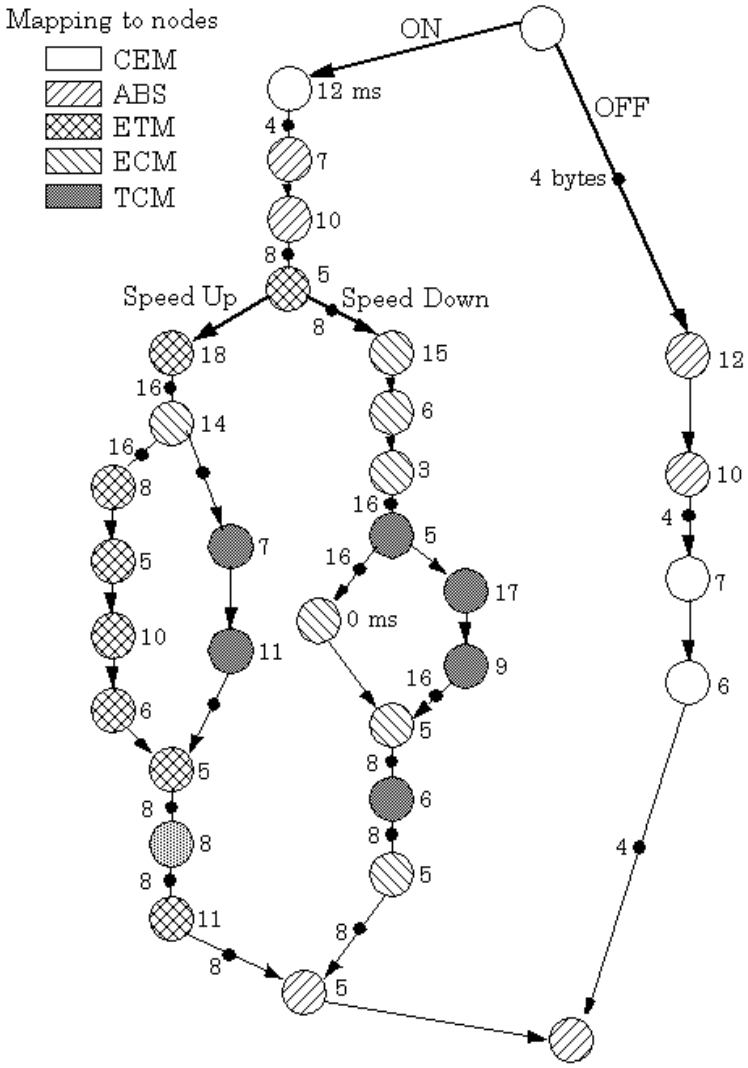


Figure 5. The cruise controller behavior

system that implements a typical safety critical application with hard real-time constraints.

The CC system allows the driver to set a particular speed and then the controller maintains that speed until the driver changes the speed, uses the brake, or switches the system off. These systems are usually controlled by a number of push buttons on the dashboard or steering wheel of the car.

In this example, we have five kinds of processes mapping to nodes functionally interact with the CC system: the Anti Blocking System (ABS), the Transmission Control Module (TCM), the Engine Control Module (ECM), the Electronic Throttle Module (ETM), and the Central Electronic Module (CEM) [23].

In Figure 5, we present the CC behaviour using a CPG. There are 32 processes in all. We have processes mapping to nodes in different

shadows. The thick lines represent the conditional edges. For different nodes, we can distinguish the processes that the nodes mapped from Figure 5, and know the types of the nodes from the definition above.

A process can be activated only if all its inputs have arrived, and when its execution finishes it transmits the information to its successors. The numbers on the right of the nodes represent the execution time of each process. And the solid circles on the edges represent the messages [15].

3.5 Petri nets

3.5.1 Petri nets Background

Petri net models are designed specifically for modeling systems when the communication, resource sharing and synchronization are important. There are four types of components: places, tokens, arcs and transitions. Places usually represent the condition, data, and resources of the petri net. Transitions usually represent actions, behaviours and events of the system. Edges connect places and transitions, and only from-transition-to-place and from-place-to-transition links exist. Tokens are used to mark the nodes. Each place can have a finite number of tokens. If each of the input places has at least one token, a transition is enabled. Transitions can be "fired" if all connected places contain tokens, one token is taken from each input place and one token is put into each output place, thus an enabled transition can be "fired" [7].

3.5.2 Petri net Example

Here we give a small example of a petri net. See Figure 6.

In Figure 6, the circles represent places (positions). The small rectangles

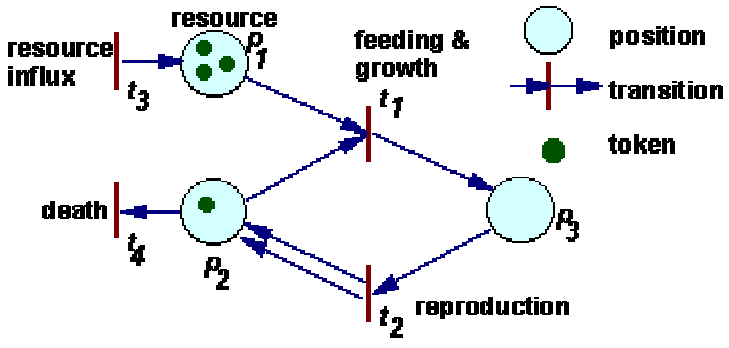


Figure 6. Example of a petri net

represent transitions.

After a transition is fired, first, tokens are taken away from places, which have arrows going from these places to the transition considered. If more than one arrow goes from place to transition, then the number of tokens removed from that place is equal to the number of arrows. Second, new tokens are placed on places indicated by arrows that originate from the transition. The number of tokens placed corresponds again to the number of arrows (in the case of multiple arrows).

In this figure we present an example process for how resources' grow and die. When transition t_1 is fired, then 1 token is removed from place p_1, 1 token is removed from place p_2, and 1 token is added on place p_3. Transition t_1 can be interpreted as feeding and growth, and transition t_2 as reproduction [26].

3.6 Summary of Representation Models

Above it was a survey about important representation models utilized in

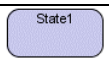
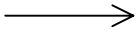
	Node Type	Edge Type	Model Constraints
UML Statechart	State  Initial State  Final State 	Transition 	•Allow self loop and multiple connections
CPG	Normal node  Disjunction node  Conjunction node 	Normal edge  Conditional edge 	•Conditional edges start from only disjunction nodes •Alternative edges from disjunction nodes meet only in conjunction nodes •No self-loop and no multiple connections
Petri net	Places  Transitions  Tokens 	Arc 	•Allow multiple connections •No self-loop

Table 1. Graph model summary

embedded systems design. We can see they share some basic similarities. For example, the elements could always be divided into nodes and edges although in different models they have different special names, and they all have some certain constraints in the relationships among the elements.

3.6.1 Models Summary

In order to have an overview about the elements and properties of these models, here in Table 1 we present the three types of models that we discussed together with their graphical elements.

3.6.2 A General Graph Model

As a conclusion from Table 1, we can see that several kinds of the graph models could be build based on a general graph model. The fundamental elements for each kind of models are nodes and edges, which differ in names, visual presentations and properties. In addition they have different relationships and inter communications due to the variant design goals of the graph models.

We can present this idea in Figure 7. As a conclusion, almost any type of graphical representation model could be created based on this structure from a general graph. This idea is used as a starting point to the design of an extensible visual graph editor.

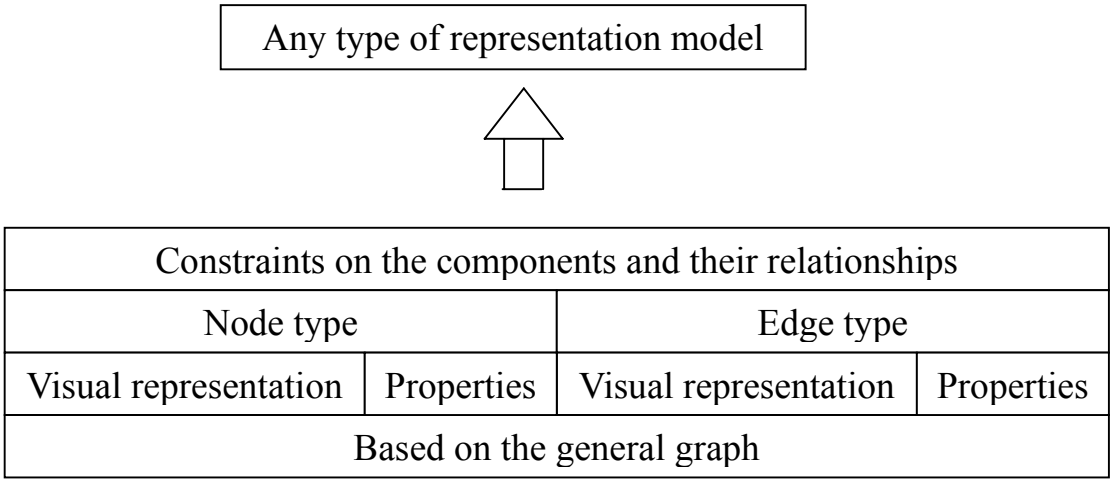


Figure 7. Graph model based on general graph

4 Related Work

4.1 Diagram Drawing/Editing Tools

Here we would briefly survey the related work about graph/diagram editors, including the editors for general graph, special editors for a particular model such as UML statechart or petri net, and graph libraries from which we could select one to extend and build an EVE.

4.1.1 General Graph Drawing/Editing Tools

Among these graph drawing/editing tools, some of them dedicate to work on a number of different general graphs. For example, Microsoft Visio, GME, Dia are some common ones.

- Microsoft Visio

Microsoft Visio could provide a series of special objects that we could use to create a UML diagram, network diagram, dataflow diagram, form, map and so on. When we select a drawing template for a special type of diagram, we could have the tools and the features for this type of diagram automatically. It is easy to find the suitable diagram that you want to create in the graph by selecting the item from menu Tools->Macros->Shape Explorer.

Figure 8 displays the interface when I am drawing the UML static structure by Microsoft Visio 2000, Standard Edition. There are complete objects including the packages, classes, interfaces and all kinds of their relationship representing edges etc. We could simply select, drag and drop the objects including the nodes and edges, and find the ports on the nodes to connect them by edges, to form a structure graph. And by double-clicking the mouse we can get the editable text field or property dialog to add the concrete attributes and information. It is quick, easy and powerful [14].

Visio can be extended using add-ons, which have to be programmed in, for example, Visual Basic.

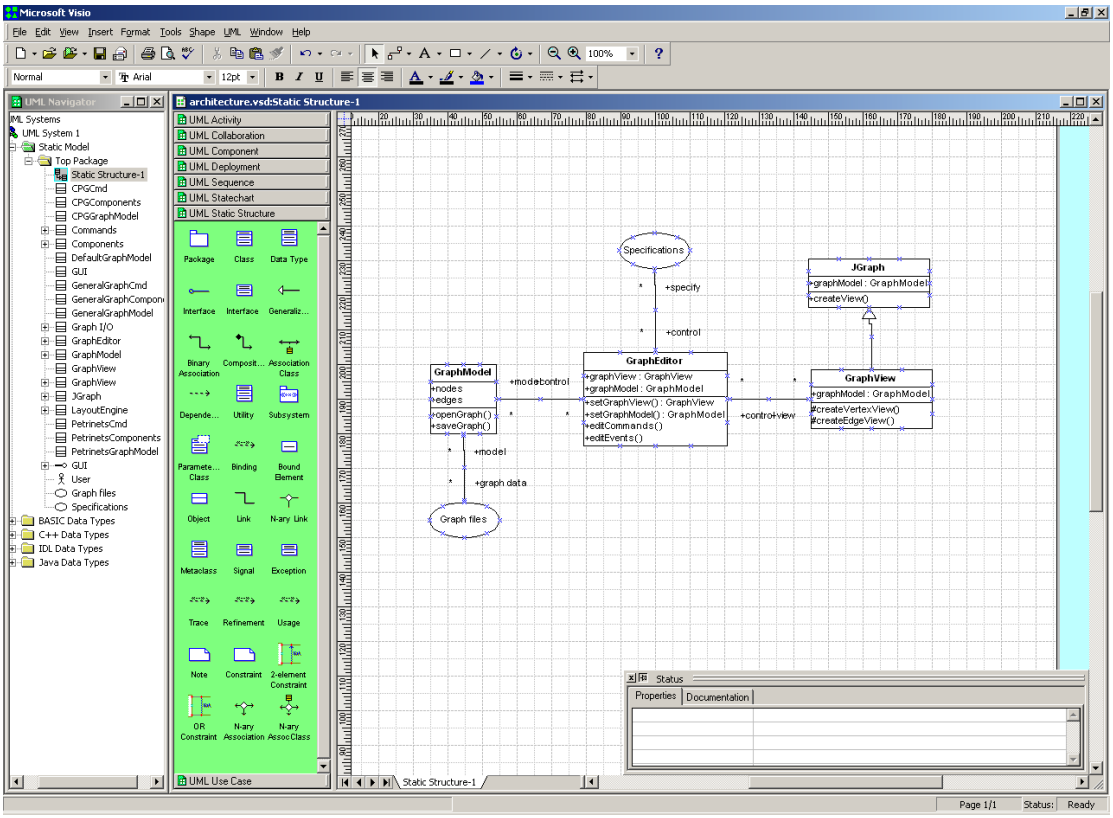


Figure 8. Microsoft visio example interface

➢ Dia

Dia is developed basing on GTK+, and GTK+ is a multi-platform toolkit for creating graphical user interfaces with a complete set of widgets [8] Dia is similar with Visio and they are both programs designed to create different types of diagrams.

Dia also contains a lot of special objects supporting the creation for many different kinds of diagrams, such as entity relationship diagrams, UML diagrams, flowcharts, network diagrams, and simple circuits.

From the tool windows in top left of Figure 9, we can start the application by creating a working window to draw a diagram. Dia provides basic functions as drawing charts, boxes and texts, and lines, and also the advanced functions as about layer, alignment, and selection and shape library.

Dia is open source software, and the source can be downloaded from [2]. There are also tutorials and manuals about Dia, which is quite easy to learn.

Dia can be extended using “shape libraries”. There are libraries for several types of graph shapes, and we can easily design our own. However, C programming is needed in order to introduce the constraints required by the model.

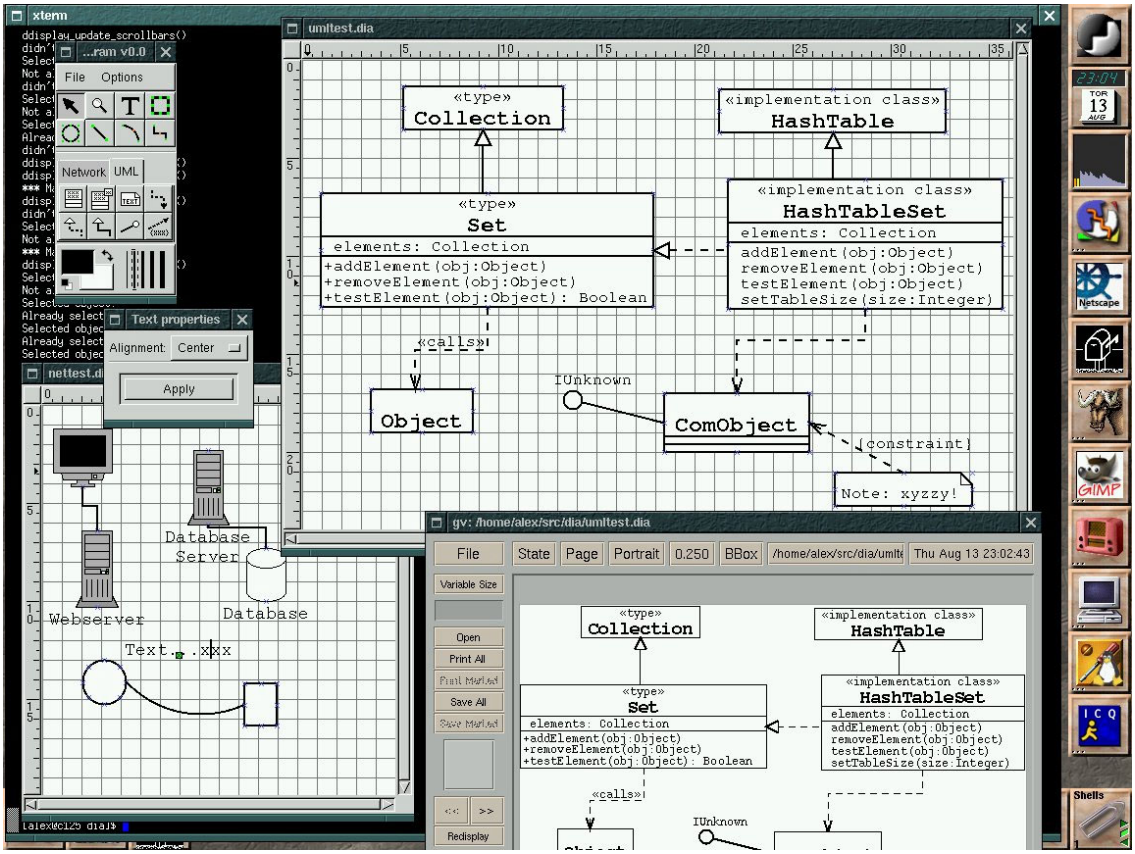


Figure 9. A full screen screenshot of Dia in action

- GME (Generic Modeling Environment)

GME is a window-based tool that provides a graphical modeling environment used primarily for model building. It supports a lot of techniques including multiple aspects, sets, references, and explicit constraints to build large-scale and complex domain-specific models.

GME gives a configurable toolkit on using meta-models specified by the modeling language. It is configurable because it can be programmed to work on a number of different domains. As for the domain, modeling language will contain all the semantic and represented information, which are used in the creation and displaying the models for the application domain.

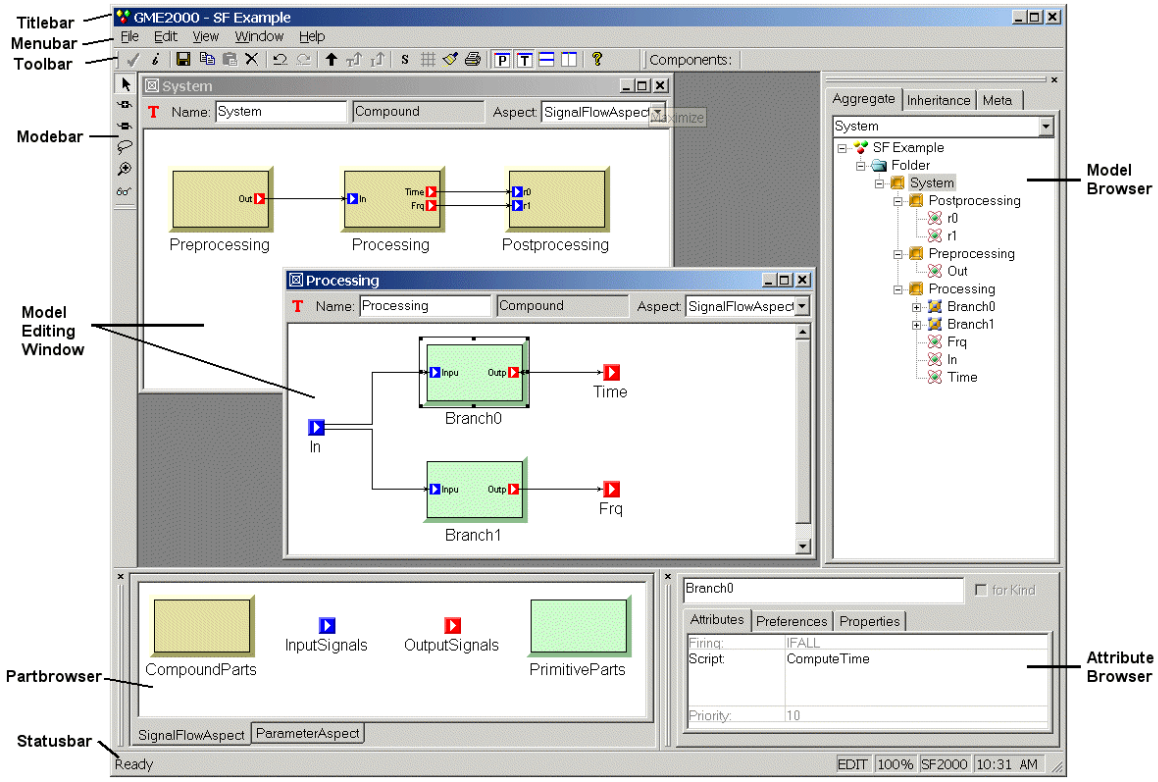


Figure 10. GME 2000 main editing window

In Figure 10 we have a look at the GME main editing window. Mainly in addition to the model-editing window, the mode bar contains buttons to select editing modes and model browser shows the hierarchy relationships or inheritance of a model. The part browser gives the parts to be inserted into the model and the attribute browser shows the attributes and preferences of an object [4].

GME is very broad, and in this thesis we were interested to develop models based on simple graph structures. It could be interesting, however, to investigate how GME can be extended with the representation models considered in this thesis.

➢ MetaEdit+

MetaEdit+ is a meta-editor that can be used to build visual modeling editors with their model analysis tools, code generators and document generators.

MetaEdit+ can be extended to handle new model types using its own meta-modeling language. For example, MetaEdit+ supports Structured Analysis and Design, which is applied to modeling techniques of data flow

diagram [13]. Figure 11 displays the interface for the data flow diagram editor. It is easy to see the function and usage clearly from the interface.

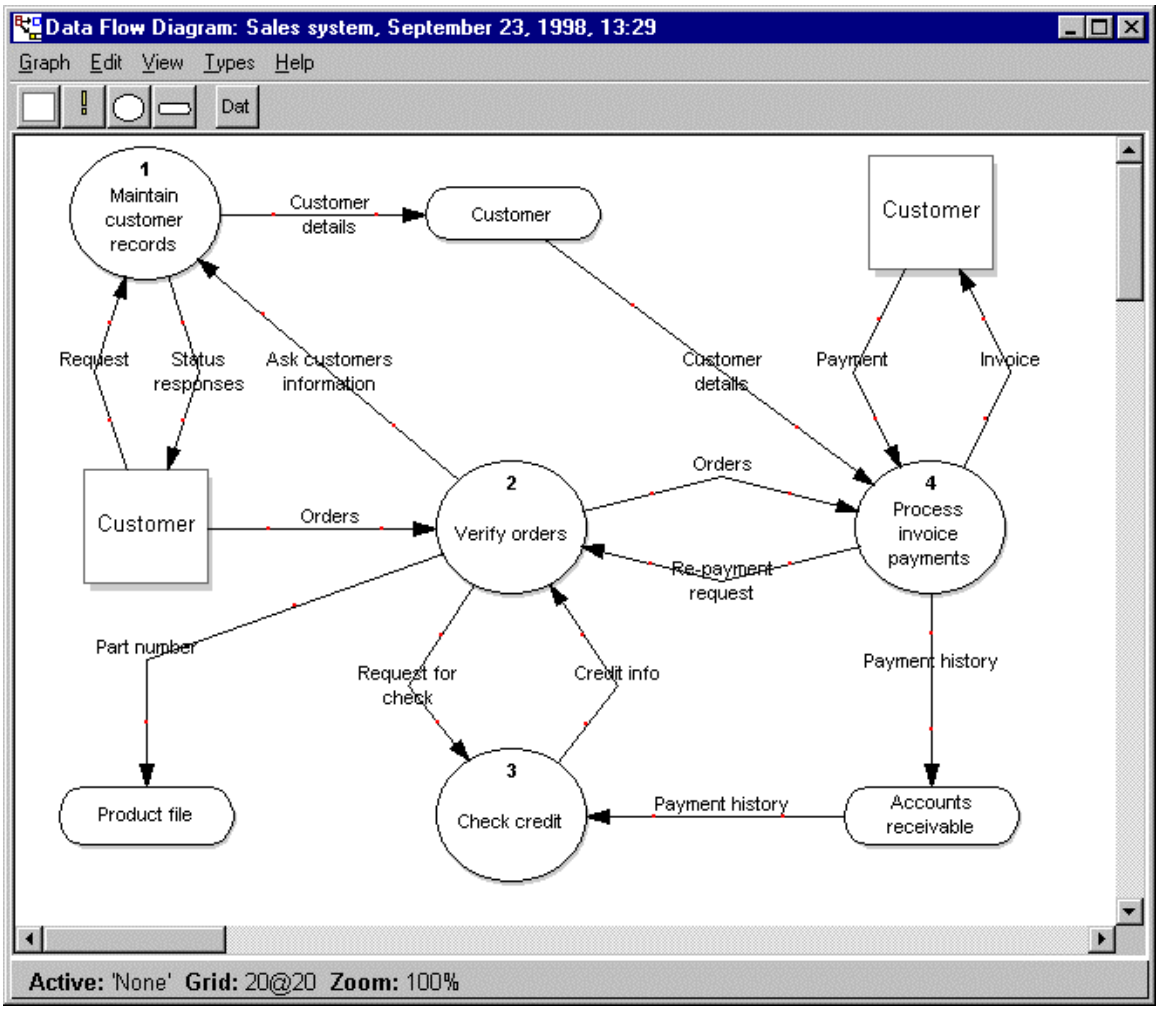


Figure 11. Data flow diagram editor

4.1.2 Tools Dedicated to a Particular Model

Among the graph drawing/editing tools, besides the editors working on general graphs, there are also some tools dedicated to a particular representation model. Here we introduce some editors being designed appropriate to edit UML statecharts, petri nets and process graphs: Statemate, PIPE, Data Flow Diagram Editor and JGraphpad.

- Statemate

Statemate is a tool dedicated to creating graphical structures and state-based behaviors through user-case diagrams, time-continuous diagrams, activity charts and UML statecharts. The statecharts can be graphically simulated and be tested whether its scenarios are correct.

Statemate has the function to generate the documents, prototype codes and test vectors automatically, and the errors that the statecharts being modeled in Statemate could be detected early in the process, this brings the advantage to reduce the cost to fix. The key components could be always reused in Statemate [21].

Figure 12 provides the interface of Statemate. A sequence diagram would be created during the simulation in UML diagrams as we can see in the interface. This sequence diagram captures the scenario that was executed, and helps to debug the model and reduce the time it takes to ensure you have captured the design intent.

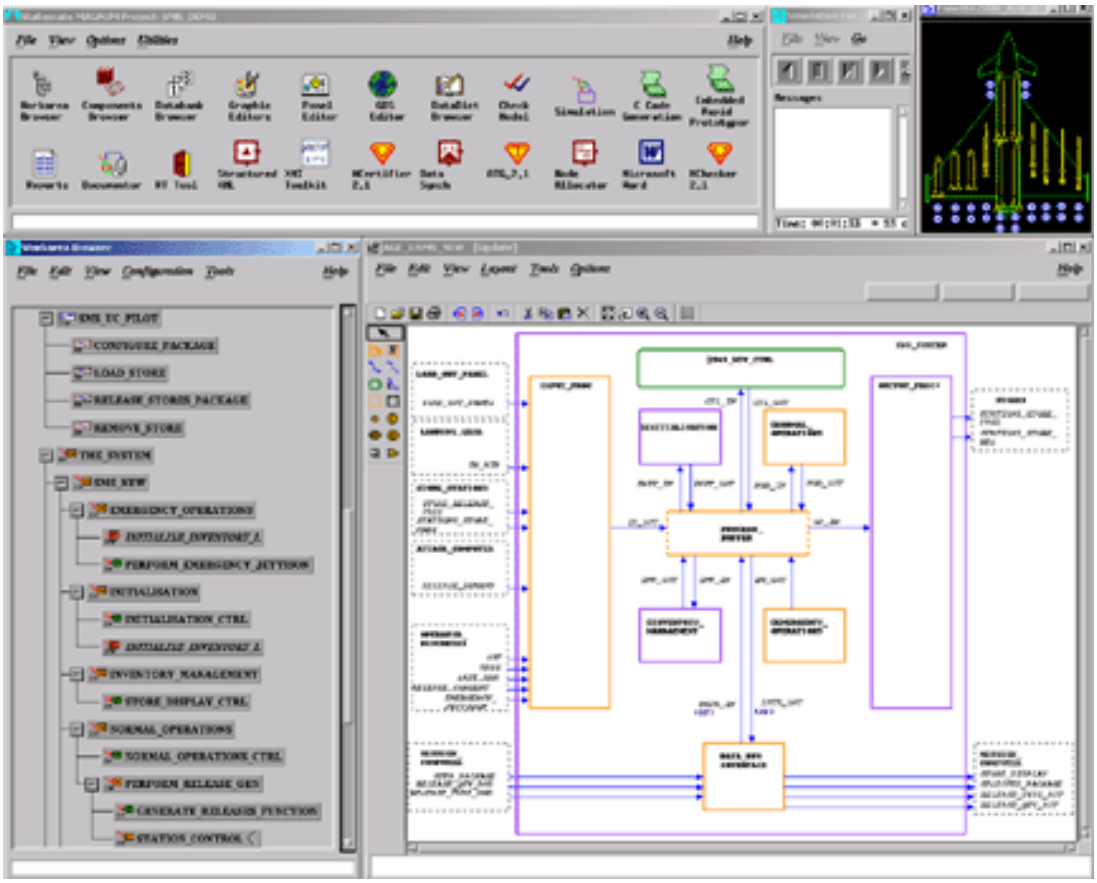


Figure 12. Statechart interface

- PIPE

PIPE is a tool designed for modeling petri nets. Using PIPE we can create a petri net, save the petri net we created, and load the petri net we created.

PIPE also has additional functionality in animation, manually firing some transitions, randomly firing a sequence of transitions, and stepping

backwards and forwards through those transitions that were fired.

The key design feature for PIPE is its extensibility: new modules can be written for checking properties of petri nets. Six analysis modules have been so far written including invariant analysis module, state-space (deadlock, etc) module, incidence-markup and enabled transitions module, simulation module, classification module and comparison module. Figure 13 presents the PIPE interface. We can see these six modules listed in the left panel. After selecting the certain module, a dialog would be created for analysis [18].

PIPE is now fully open source.

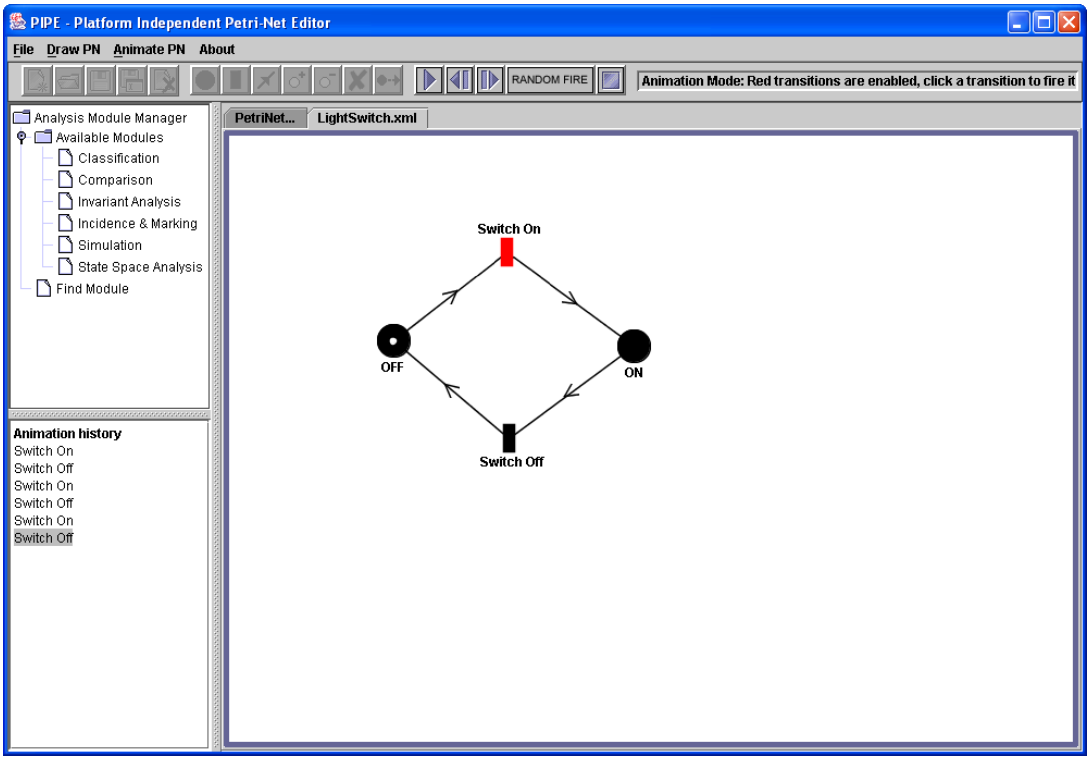


Figure 13. PIPE interface

- JGraphpad

JGraphpad is a free diagram editor based on JGraph, to create flow charts, maps, UML diagrams, and so on. JGraphpad is provided as an example for the JGraph Swing component. JGraphpad could be a versatile product that is used to display and edit any type of diagram in software engineering, transport network and workflow systems.

The main features of JGraphpad are that it has programmable tool bar, GUI and points that we could creates our own tool bar buttons, and it supports a big range of platforms including Windows, Linux, Mc and Solaris.

If the process graph is inputted in text file, the imported form could be GXL, JPG, PNG, HTML Image maps. JGraphpad is free software [11].

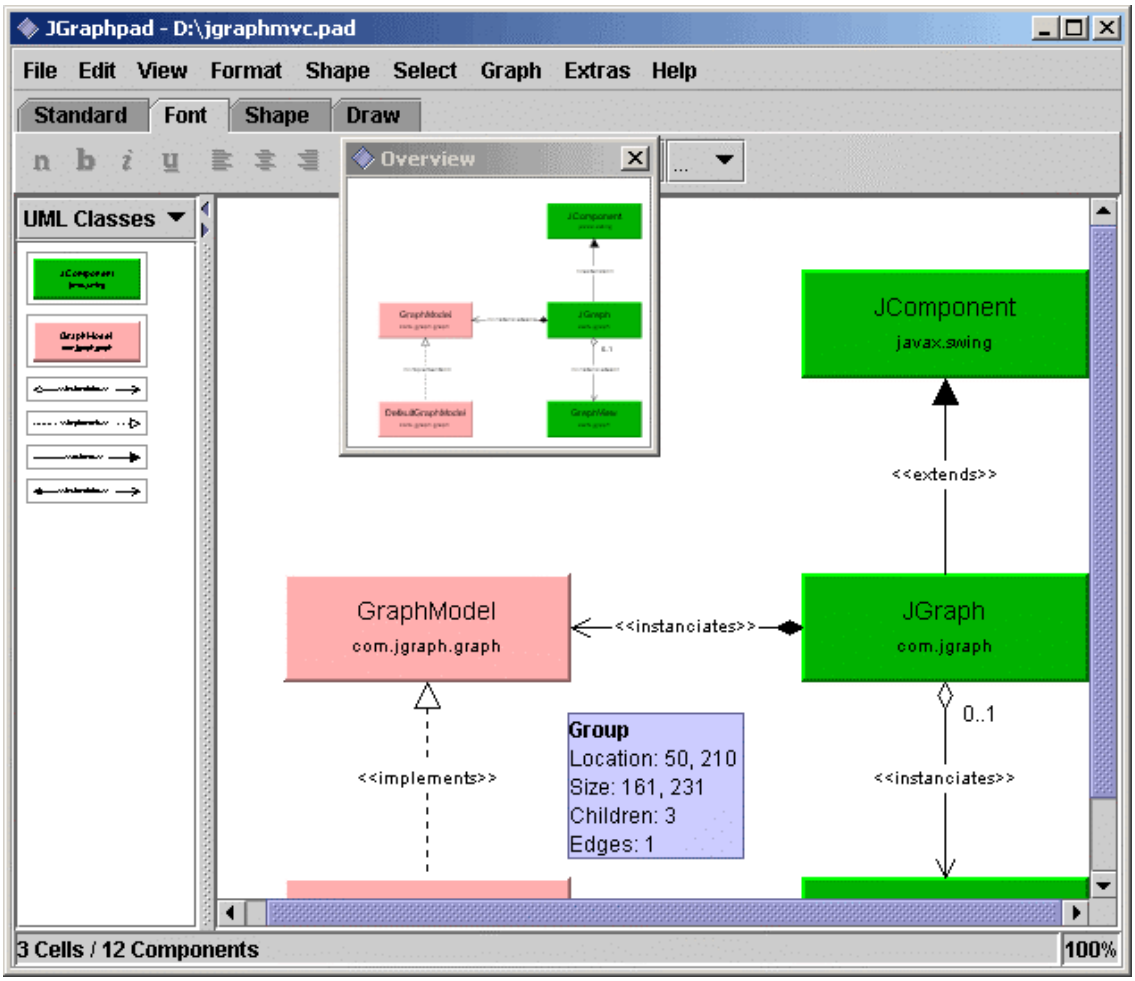


Figure 14. UML in the metal look-and-feel

Figure 14 presents a UML diagram drawn to show MVC structure in a sysytem. We mentioned here JGrahpad is because it is based on the graph library JGraph, which is utilized by the EVE we designed in this thesis.

JGraphpad can be extended by programming in Java.

4.2 Graph Libraries: GEF and JGraph

There are several libraries available for graph editing. For example, GEF

is a library of Java classes for editing diagrams and connected graphs, and ArgoUML is successful software built on GEF as a UML editor. JGraph is a graph library in pure Java for editing all kinds of graphs, and JGraphpad is a complex editor dealing with flow charts, maps, UML diagrams, and so on, which is built on JGraph.

4.2.1 Graph Editing Framework (GEF)

The goal of GEF is to build a library of Java classes to be used to construct many, high quality connected graph-editing applications. The basic GEF functionality is something like Visio, which gives a core functionality ability to drag-and-drop diagram objects onto the diagram and then creates links between them. Right now it can read and write PGML (Precision Graphics Mark-up Language) [17] files, but later it might be updated to use SVG (Scalable Vector Graphics) [22] instead. Here is the demo implementation of GEF in Figure 15.

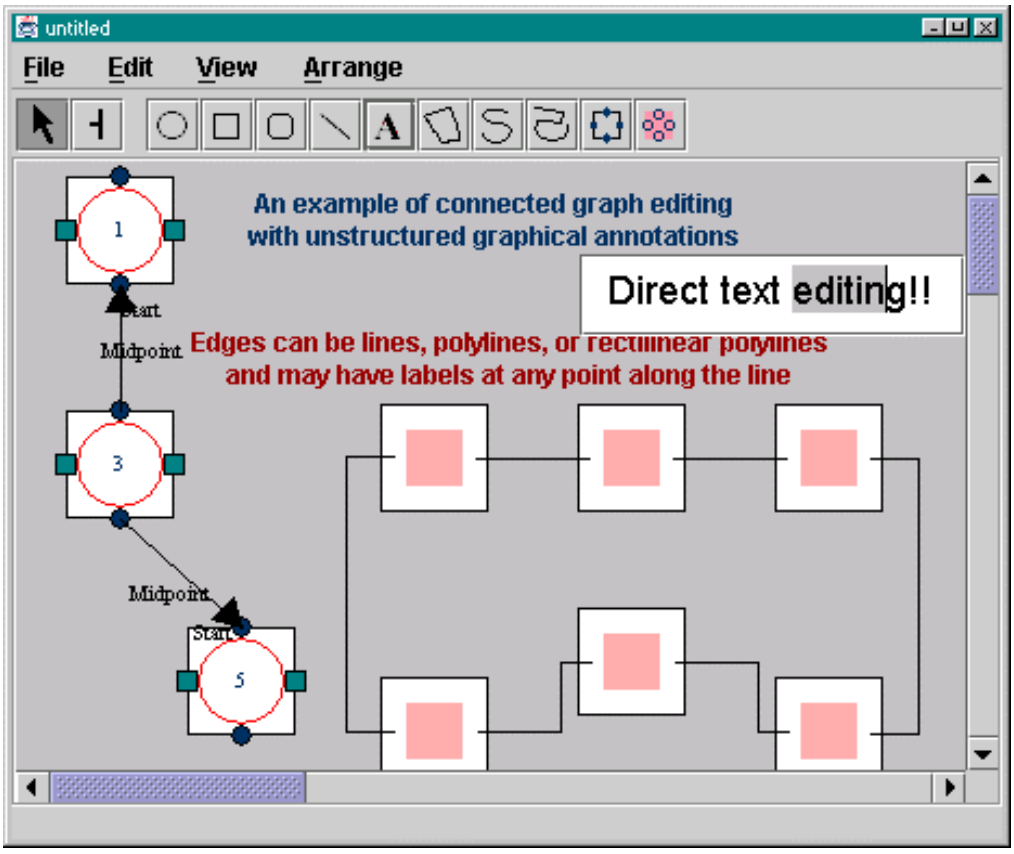


Figure 15. Example demo of GEF

Through the interface users can edit the connected graph in a visual way. GEF uses a node-port-edge model, and there are two types of nodes as displayed in this example demo despites the geometric shapes in the tool

bar. The edges can be lines, polylines and rectilinear polylines. Ports are fixed on the nodes in two deferent types. An edge can connect only same-type ports. For example, in the previous figure, if the edges are created between the ports represented as two black dots, only the polyline could connect with a black arrow. The multiple output or input edges of a port is not allowed.

Concretely GEF covers these features as we conclude in Table 2 [6].

Operations	**Functionalities**
Tools	Tools like selection, creation, connection and marquee
Palette	A palette for displaying those tools
Size	Handles for resizing objects and bending connections
Commands	Multi-Undo/Redo support and multi-activation sites
Controller	A controller framework for mapping the model to a view
In-Place Edit	Direct text editing
View	Two types of GEF viewers—Graph and Tree
Access	Keyboard navigation and in future, text to speech
Drag and Drop	Native drag and drop support
Align	Alignment actions

Table 2. Main Features of GEF

Figure 16 presents a general view of the packages in GEF library.

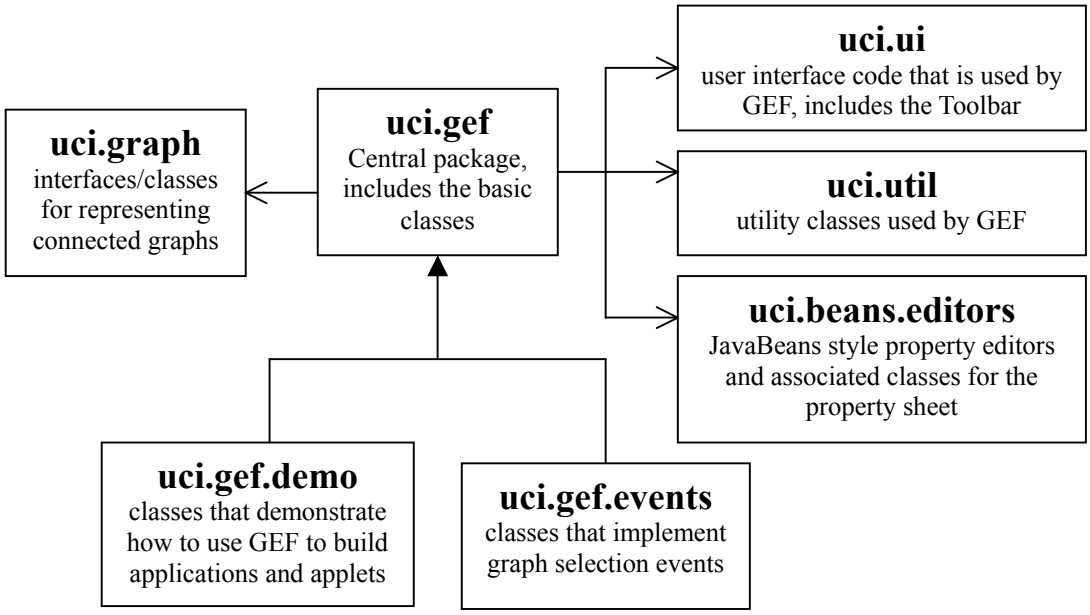


Figure 16. GEF packages

We first look into package uci.gef since it is the biggest and central

package of GEF including most primary classes in GEF, such as JGraph, Editor, Fig, Selection, Layer, Guide, Mode, Cmd, and NetPrimitive.

More detailed, Class JGraph (notice this is not the JGraph library presented in the next section!) extends javax.swing and contributes to display a connected graph and allow interactive editing. It could be considered as a simple front-end to class Editor. Class Editor provides an editor for manipulating graphical documents, without the need to contain much code because all the graphical objects, layers, editor modes, editor commands, and supporting dialogs and frames are implemented in their own classes. Cmd performs the actions in Editor. Modes are modes of operation for the Editor that interpret user input and instantiate Actions. Guide constrains user mouse coordinates to help make an organized looking diagram. Fig contains a lot of draw-able objects including lines, rectangles and circles. Layers contain the objects to be drawn. Selections are objects used by the Editor when the user selects a Fig that indicates the target of the next Action. NetPrimitive is the parent class of all Nodes, Ports, and Arcs.

In addition, package uci.gef.event contains classes that implement graph selection events. Package uci.gef.demo contains classes that demonstrate how to use GEF to build applications and applets together with the HTML files for the demos. Package uci.graph contains interfaces and default classes for representing connected graphs. We can create our own application-specific objects to represent connected graphs by GEF like in Swing, as long as we implement a GraphModel to let GEF access to our objects. Package uci.ui contains user interface code that is used by GEF but could also be used for other purposes, like Toolbar. Package uci.util contains utility classes that are used by GEF but could also be used for other purposes, like a progress bar window. Package uci.beans.editors contains JavaBeans style property editors and associated classes for the property sheet [5].

4.2.2 JGraph

JGraph is a library of pure Java classes developed for editing different types of graphs including UML diagrams, maps, flowcharts, network diagrams and so on. JGraph supports to drag and drop the selection modes and display/edit options for editing the graph as well as GEF. JGraph could read a GXL (Graph eXchange Language) [8] graph, apply a custom layout algorithm, and return the result as an SVG (Scalable Vector Graphics) image.

A simple example of a program implemented using JGraph library is presented in Figure 17. This program provides a simple diagram editor, which allows connecting nodes by means of connecting the ports. It is similar to GEF in way of node-port-edge design. But the obvious difference is that one node only has one port, and the port that binds to a node, is a floating one. And there is no constraint on the number of input or output edges connected with the port.

If we want to create a graph, we simply drag and drop diagram objects onto the graph pane and create links between them, in JGraph example, nodes and edges have editable text field themselves which we can enter short information in. These are no different types of nodes or edges in this simple example, but in JGraph it is easy to deploy from the library to create variant nodes and edges with special shapes and graphical attributes, as long as we provide the special rendering methods. And in toolbar there are also commands like delete, cut, copy, paste, undo, redo and so on. In this simple example there is no menu but of course we can create one depending on the library.

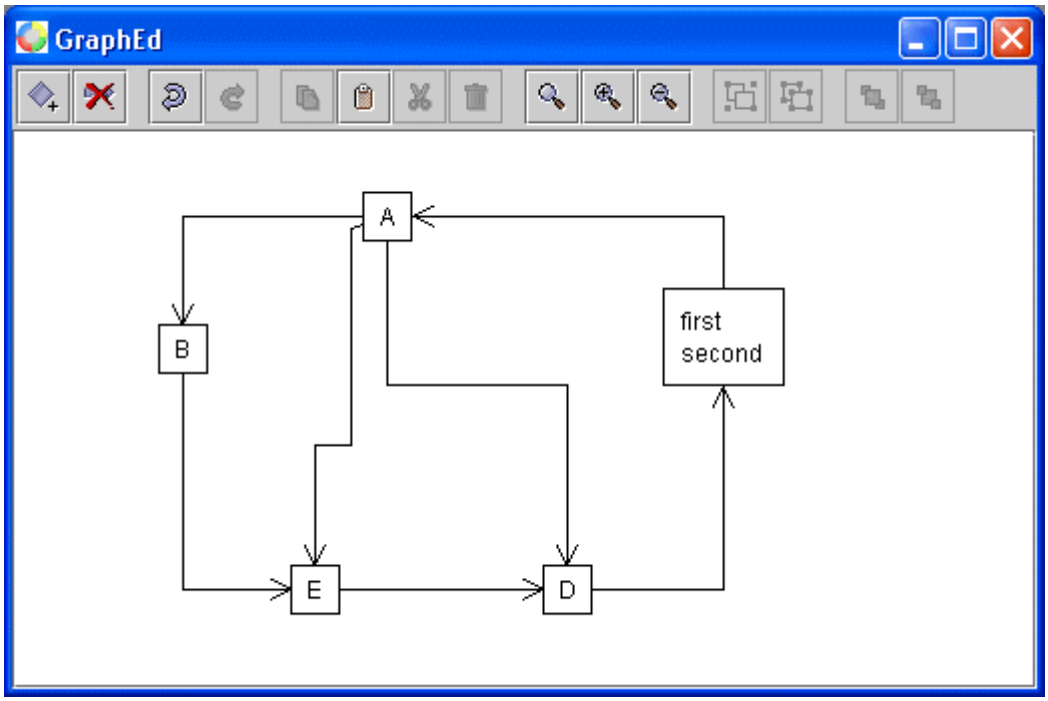


Figure 17. An example of JGraph usage: simple diagram editor

Concretely JGraph has the following functionality:

Operations	**Functionalities**
Edge Editing	Add/Remove/Edit Points; Connect, Disconnect; Labels
Moving/Sizing	Transaction-Based, with Live-Preview
Selection	Single-Cell and Rubber-band Selection
Zoom	Arbitrary Zoom; Uses Java2D
Layering	View-Dependent Inter- and Intracell Layering
Grouping	Children Selectable; Uses Tree-Interface
Grid	Customizable Size, Color, Appearance
In-Place Editing	Direct Text Editing for all Cells
View Attributes	Separate Attributes for each attached View
Graph Layout	Easy Integration of Custom Algorithms
Ports	Floating Connection Points for Vertices
Handles	Flexible Interface for Cell-Modifications
Drag and Drop	Between JGraphs, JVMs and other applications/OS
Clipboard	Supports Multiple Transfer Formats
Command History	Multi-View; for all available Operations
Look-and-Feel	All Swing Pluggable Look-and-Feels
Routing	Customizable Routing with Default Algorithms
Visibility	Hide edges, vertices and groups
Clustering	Folding/Unfolding of Groups into Vertices

Table 3. Main features of JGraph

JGraph is based on the Model-View-Controller (MVC) design pattern (see section 5.2.1). The following are the main packages in JGraph library. All classes in JGraph have their equivalents in Swing, and all features are fully standards-compliant.

Figure 18 presents the architecture of JGraph library in UML [11].

Package com.jgraph is JGraph's topmost package. It contains the JGraph class. Class JGraph is a control displaying related objects of a graph, and extends JComponent. It has a reference to its GraphUI and GraphModel. Its object doesn't hold data but simply provides a view of the data. The graph gets data by querying its data model. Package com.jgraph.plaf.basic contain class BasicGraphUI that extends GraphUI, which in turn extends ComponentUI. They act as control part in MVC in JGraph. Package com.jgraph.event contains event classes and listener interfaces. Package com.jgraph.graph defines a number of classes and interfaces and provides support classes that include the graph model, graph cells, controllers, and renderers and so on.

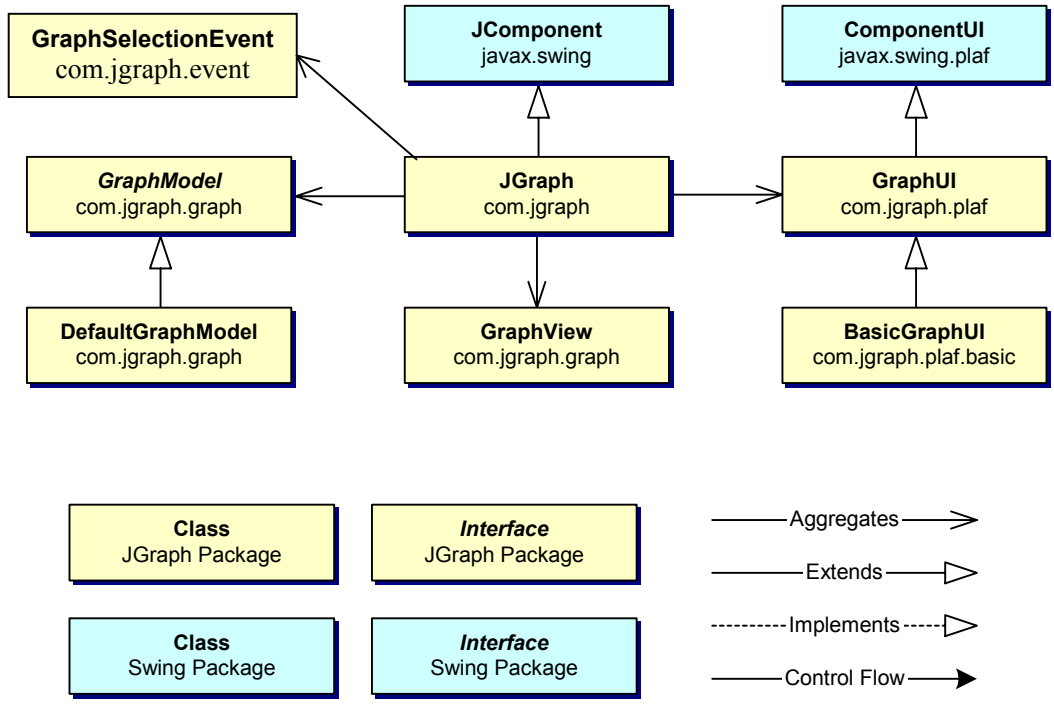


Figure 18. JGraph MVC in UML

In order to understand clearly node-port-edge graph model of JGraph, we still have to look into the inter structure of package com.jgraph.graph to know how the classes concerned with the cells work in JGraph.

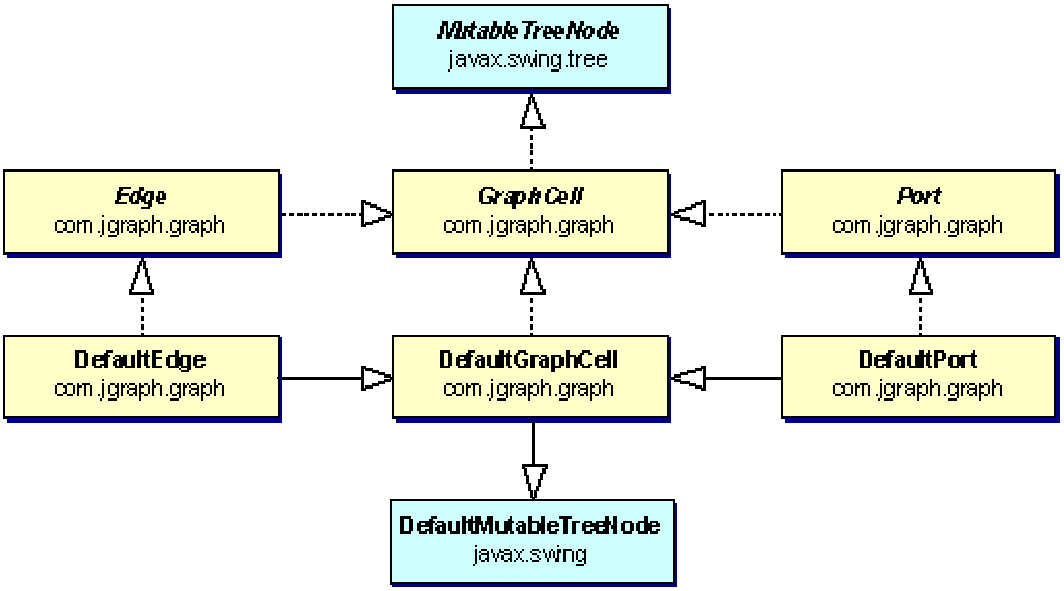


Figure 19. GraphCell interface hierarchy and default implementations

As we see from Figure 19, the GraphCell interface hierarchy and default implementations, DefaultPort and DefaultEdge, are respectively simple implementations for a port and an edge. They extend DefaultGraphCell, which gives a default implementation for the GraphCell interface. And GraphCell is the basic interface for all graph cells and it defines the requirements for objects that appear as cells. Port and Edge are respective interfaces providing the definitions on the requirements for an object that represents a port or an edge in a graph model.

4.2.3 Comparison of GEF and JGraph

Both GEF and JGraph are powerful graph libraries applied to build graph-editing applications with a lot of functionality features, and they each have some advantages and disadvantages. In the following we present a comparison between them.

- Advantages

In GEF, the main objects are very concrete and familiar if we have experience in using drawing tools. It doesn't contain many abstract concepts such as constraints or event handlers like some other libraries.

GEF is adaptable to a wide range of applications, and is efficient on diagrams up to a thousand elements with large numbers of features, thus it has good scalability and can handle very complex types of diagrams.

In JGraph, there is a simple API that is similar to standard Swing components, so it is easy to learn and to deploy JGraph. The existing source code in JGraph can be reused, which make the development time shorter. With JGraph, we from highly expert to very non-expert are able to display and edit complex graph models without the need to understand the underlying complexity.

JGraph is dedicated to graphs instead of complex diagrams, and this is also the focus of the thesis. JGraph has also well written documentations and rich examples, so it is much easier and faster to learn.

Both GEF and JGraph are open source software.

- Disadvantages

In GEF library there are already 100 classes, which makes it harder to understand the whole construction of the system. And since there are too

many features provided by GEF, a lot new features are not properly explained. GEF doesn't have enough examples and well-written documentations either, which increases the difficulty for a fresh user to learn.

JGraph uses non-Swing API for more advanced features such as layering, grouping, cloning, ports, and they are not used everywhere in standard Swing, and then it requires new classes and methods which increases the complexity.

Both GEF and JGraph contain bugs. For example, in GEF, FigEdgeRectilinear should move when both their end points move at the same time, but some points on the arc stay without moved, and in category of view of JGraph, when using groups that contain both cells and ports, the edges attached to these ports are not properly updated.

4.2.4 JGraph Library Example

Since JGraph is easy to deploy and learn with well-written documents and API, and with floating ports, multiple routes in connection, and editable text fields in the nodes, JGraph is easier and more suitable to be applied to process graph editing. Thus, in the next chapter, we would implement a prototype of CPG Editor based on JGraph.

Before that, let's look at a simple JGraph library example. First the following is the code "Hello.java" to create a window displaying a very basic simple graph with only two nodes and an edge.

```
import com.jgraph.*;
import com.jgraph.graph.*;
import javax.swing.*;
import java.util.*;
import java.awt.*;

public class HelloWorld {
   public static void main(String[] args) {

      // Construct Model and Graph
      GraphModel model = new DefaultGraphModel();
      JGraph graph = new JGraph(model);
      graph.setSelectNewCells(true);

      // Create Nested Map (from Cells to Attributes)
      Map attributes = new Hashtable();
```

```
// Create Hello Vertex
DefaultGraphCell hello = new DefaultGraphCell("Hello");

// Create Hello Vertex Attributes
Map helloAttrib = GraphConstants.createMap();
attributes.put(hello, helloAttrib);
// Set bounds
Rectangle helloBounds = new Rectangle(20, 20, 40, 20);
GraphConstants.setBounds(helloAttrib, helloBounds);
// Set black border
GraphConstants.setBorderColor(helloAttrib, Color.black);

// Add a Port
DefaultPort hp = new DefaultPort();
hello.add(hp);

// Create World Vertex
DefaultGraphCell world = new DefaultGraphCell("World");

// Create World Vertex Attributes
Map worldAttrib = GraphConstants.createMap();
attributes.put(world, worldAttrib);
// Set bounds
Rectangle worldBounds= new Rectangle(140, 140, 40, 20);
GraphConstants.setBounds(worldAttrib , worldBounds);
// Set fill color
GraphConstants.setBackground(worldAttrib, Color.orange);
GraphConstants.setOpaque(worldAttrib, true);
// Set raised border
GraphConstants.setBorder(worldAttrib,BorderFactory.createRaisedBevelBorder());

// Add a Port
DefaultPort wp = new DefaultPort();
world.add(wp);

// Create Edge
DefaultEdge edge = new DefaultEdge();

// Create Edge Attributes
Map edgeAttrib = GraphConstants.createMap();
attributes.put(edge, edgeAttrib);
// Set Arrow
int arrow = GraphConstants.ARROW_CLASSIC;
GraphConstants.setLineEnd(edgeAttrib , arrow);
GraphConstants.setEndFill(edgeAttrib, true);

// Connect Edge
ConnectionSet cs = new ConnectionSet(edge, hp, wp);
Object[] cells = new Object[]{edge, hello, world};
```

```
        // Insert into Model
        model.insert(cells, attributes, cs, null, null);

        // Show in Frame
        JFrame frame = new JFrame();
        frame.getContentPane().add(new JScrollPane(graph));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

The previous code creates two nodes, an edge to connect the nodes and inserts all the cells in the model, using the classes we have explained in Figure 19.

In the main part of the code, we create the node “Hello” and define its position and size. Then the map “helloAttrib” is created from the GraphConstants to hold the properties. We put the node and its properties in a general hashtable called “attributes” respectively as the key and the value. A port is also added into the node. Through the similar way we create the other node “world” and the edge connecting them. ConnectionSet represent the set of connections, so we connect the two ports with the edge by creating an object called “cs”. The array called “cells” contains all the elements. Finally, we insert these elements into the graph pane. Figure 20 shows the resulting graph when run the completed application.

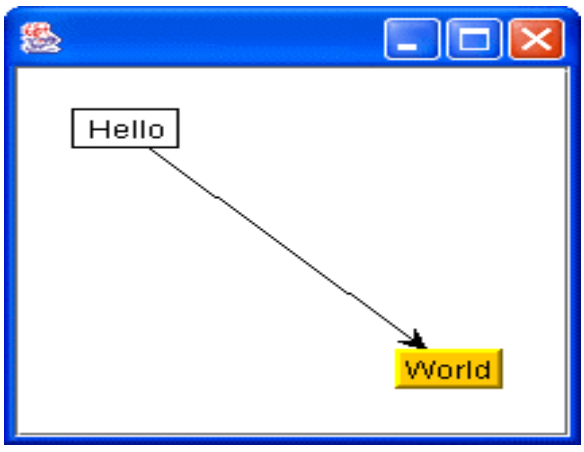


Figure 20. Result window of “Hello.java”

5 A Prototype Implementation of a CPG Editor

5.1 Features of a CPG editor

In order to understand the issues involved in building an extensible visual editor, we have built first an editor for conditional process graphs.

Our objective is to produce a tool that supports the creation and editing of a CPG. With the editor, we can create a new CPG that has its own set of nodes and edges with their separate attributes. The editor is responsible to display a CPG model whenever we open a correctly defined CPG file (with the suffix name .cpg), and to save a CPG to a file.

The elements of a CPG are have been presented in Table 1. There are three node shapes--circle, triangle and inverse triangle--representing the three types of nodes. The shapes contain labels with the nodes' name, resource and the worst-case execution time. There are two kinds of edges: normal edges and conditional edges, whose ends are respectively represented by thin and thick directed arrows.

Figure 21 presents the user interface for the CPG editor. There are mainly three parts: menu, toolbar and graph panel.

1. Menu "File" has items New, Open, Close, Save, Save as, Page Setup, Print Preview, Print and Exit. Menu "Editor" contains items Undo, Redo, Cut, Copy, Paste and Select All;
2. Menu "View" contains items Options Zoom in and Zoom out; Menu "Insert" contains items Process, Conditional process and Conjunction process;
3. "Help" contains item About.

The above commands are common almost in any graph editor, except the three ones designed to insert CPG nodes. In the tool bar we provide buttons as shortcuts of the commands in the menu.

After opening a certain CPG file, a CPG model is created in the graph panel and user can move, add, delete the cell, and then save the changes in the CPG file. We mainly implement the basic function as modifying the cells' attributes on names, resources and worst-case execution time and so on.

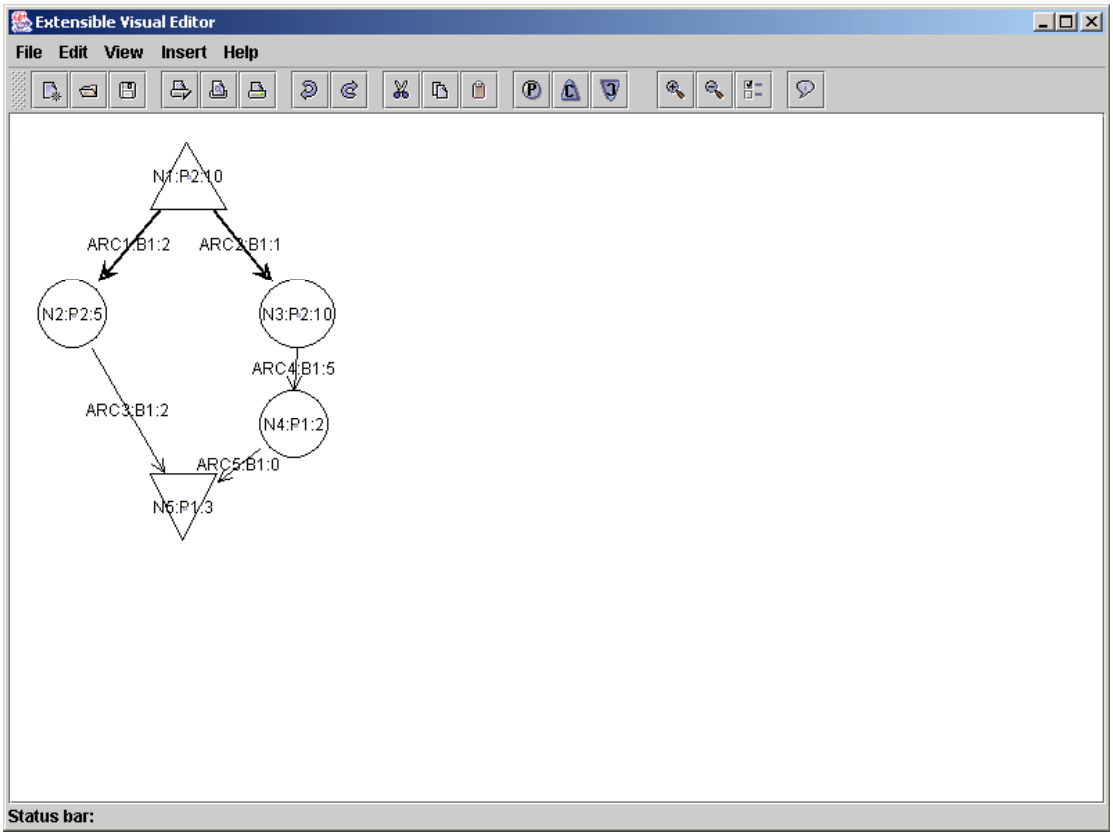


Figure 21. User interface of the CPG editor

This CPG editor operates on the CPG files with the suffix name .cpg. Currently, to simplify the application, we use a simple textual format. In the future, CPG files could be represented using XML. The .cpg file records the cell attributes. For nodes it includes node type, name, x coordinate axis position, y coordinate axis position, worst-case execution time, resource (the process that the node maps) and condition. For edges it includes edge type, name, name of source node, name of destination node, worst execution time, bus and condition.

Here it is an example of CPG file named example.cpg. Each item of the attributes is separated by a blank space. We can indirectly create a new CPG model by writing a CPG file.

example.cpg
disjunction_node N1 100 20 10 P2 C
node N2 20 120 5 P2 null
node N3 180 120 10 P2 null
node N4 180 200 2 P1 null
conjunction_node N5 100 260 3 P1 null
conditional_edge ARC1 N1 N2 2 B1 C
conditional_edge ARC2 N1 N3 1 B1 -C

edge ARC3 N2 N5 2 B1 null
edge ARC4 N3 N4 5 B1 null
edge ARC5 N4 N5 0 B1 null

We can create a new CPG by inserting different types of nodes and connecting them with the edges, also we can give each cell including nodes and edges its attributes by typing short information in their editable text field and pressing “enter”. Because there are only two types of edges in CPG, it is easier to create the edges by dragging and releasing the mouse from source node to destination node. We have set the constraints about the connection that each edge from a disjunction node is always a conditional edge in the application. From reading the type of the source node, different types of edges are created automatically.

A CPG can also be modified (edited). We can modify the names, execution times, and resources of the cells by putting the mouse in the selected cell and double-clicking the mouse, then we get the editable text field that we can type the new attributes, then press “enter”, in this way the model saves the modification, when we click command button “save”, all the changes are finally saved back to the CPG file. Cells can also be moved, cut, pasted and copied, but currently we didn’t implement other edit functions.

5.2 The Design of the CPG Editor

5.2.1 Model-View-Control (MVC) Architecture

MVC is widely used in implementing applications that designed for graph drawing or editing with graphical user interfaces. It means that the framework could be split into three parts: model, view and control.

The model part is the content representation of the system. It describes the underlying graph model interface, and selection model, and the elements that they contain, as well as the classes used to change the graph model.

The view part is the graphical representation of the system. It displays the graph represented by the model. Mainly it focuses on the geometric shapes or images of the graph elements being displayed.

The control part is a manager that mediates and communicates between the model and the view, it explains the graph rendering process, provides the model constraints, and shows the interaction with the graph model through the interface [6].

5.2.2 The Architecture of the CPG Editor

In Figure 22 we present a UML diagram indicating the structure of the application. We can see that the MVC design pattern is used. Most classes in the library extend from JGraph, such as CPGModel, CPGView, EllipseView, EllipseCell.

There's no "model specification file" for the prototype CPG editor currently since it provides only operations especially on CPG. We do not need to tell the application what to do according to different model specifications. Actually we simply transfer the information that should be included in the specification to the programming codes given if-else instructions, and then the editor knows what to do when meeting with different types of nodes and edges. But in the case of the extensible visual editor we have to define specifications for each type of graph.

The CPG editor uses the MVC design pattern, so the software structure has to be split into three parts: model, view and control:

- The model part provided by class CPGModel, describes the underlying CPG model interface, and selection model, and holds the content of three types of nodes together with two types of edges that a cpgModel contains.
- The view part provided by class CPGView, studies the display's internal representation of a CPG, and the mapping and update between the cpgModel and the cpgView.
- The control part provided by the CPGEditor, manages the cpg rendering and the interaction with the cpgModel through the interface.

CPGModel extends DefaultGraphModel. It gives the implementation of a CPG model, which in this editor is called cpgModel, and a cpgModel has objects respectively of EllipseCell for normal node, TriangleCell for disjunction node and InverseTriangleCell for conjunction node. All of them extend DefaultGraphCell and provide an implementation of GraphCell interface, which defines the requirements for objects that appear as graph cells.

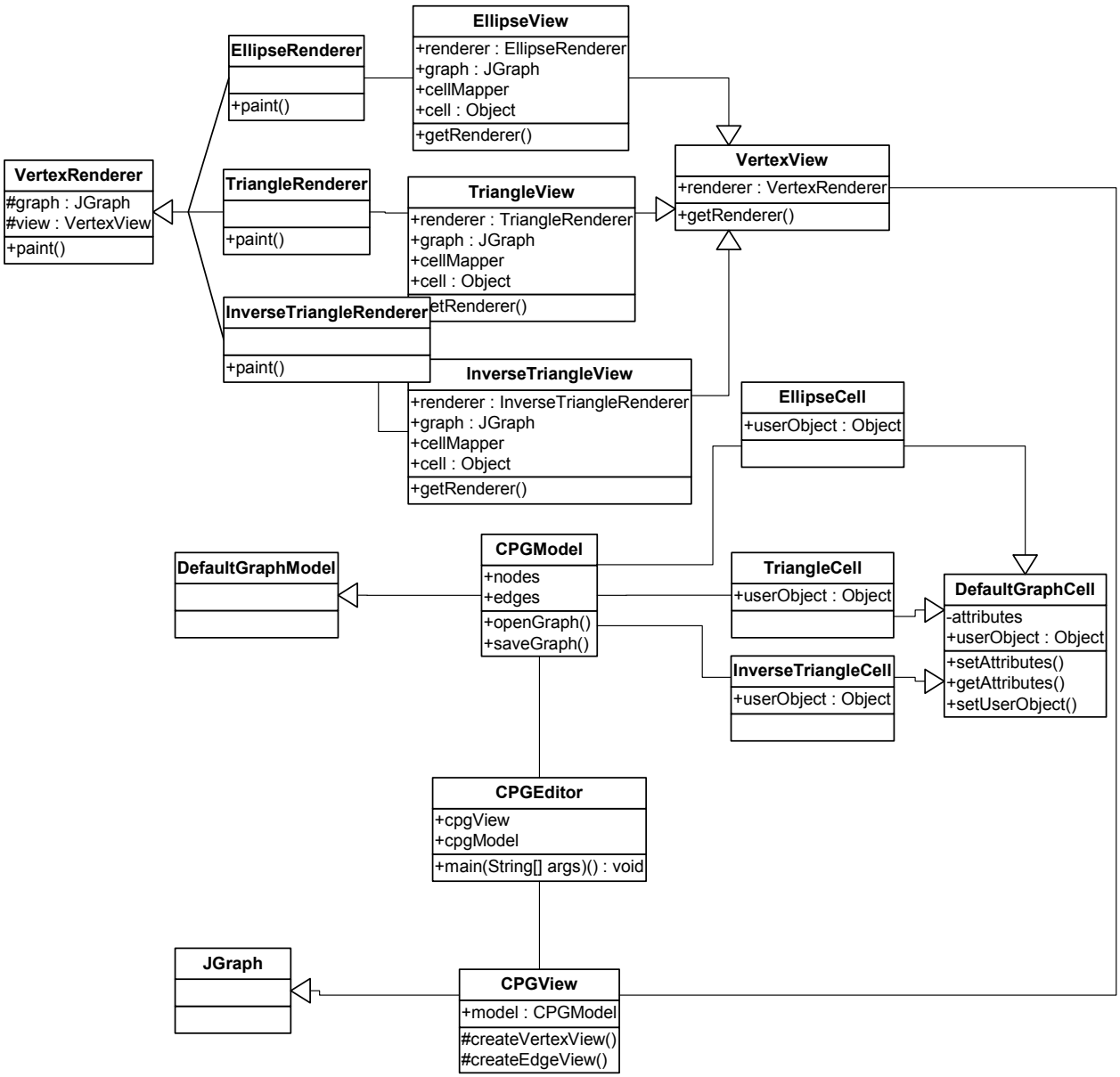


Figure 22. UML class diagram for CPG editor

CPGView extends JGraph, it displays the objects in a CPG. It doesn't actually contain the CPG data and it simply provides a view of the CPG data. The CPG gets data by querying its data model, cpgModel. CPGView displays the data by drawing individual cells of CPG. Method CreateVertexView draws the nodes, including EllipseView for normal node, TriangleView for disjunction node and InverseTriangleView for conjunction node with their respective renderer to draw the shapes. These three view classes all extend VertexView. Method CreateEdgeView creates the edges. Both VertexView and EdgeView implement interface CellView, which defines the requirements for an object that represents a view for a CPG model cell.

In order to connect a pair of nodes successfully, the edges are created on the ports that bind with the nodes. In this CPG editor we simply use the floating port in JGraph and import the classes PortView to draw the port and DefaultPort to create the port cell.

5.3 Implementation

The code was developed in Java 2 Platform Standard Edition (J2SE) 1.4.0 [10]. The CPG editor is developed using the JGraph library. We import these three JAR files from the project: jgraph-2.1-java 1.4/jgraph.jar (now the new version turns to jgraph 2.2.2), jgraph-1.0.6-java1.4/jgraph.jar and jgraphpad-2.0.0/jgraphpad.jar (now the new version is jgraphpad 2.2.2.1), as the required libraries. And we have the basis classes to extend our own Java files, seen from JGraph v2.2.2 API Specification [11].

In CPGModel we create two classes CPGNode and CPGEdge to hold all the attributes for nodes and edges respectively. The information recorded inside a CPG file are fetched out to open a new CPG from the CPG file, Classes CPGNode and CPGEdge are used to hold the attributes data of cpg cells. CPGNode objects are created and put into a hash table named nodes. CPGEdge objects are created and put into a linked list named edges. “nodes” and “edges” are associated by setting names of the source nodes in edges as the key. So in the hashtable “nodes”, the key is the node name and the value is the node itself.

There are the instructions in the codes to go through both the hashtable nodes and the linkedlist edges, and get the cell information from cpgModel. Reading certain CPG node type and edge type and other attributes obtained from the CPG file. CPGView creates the corresponding views of CPG cells by methods createVertexView and createEdgeView. Method openGraph in CPGModel inserts all these cells with their attributes and their relationships into a CPG editor pane.

Then we edit on the CPG such as to change the attributes, add a normal node, a disjunction node or a conjunction node, and find the ports to connect any pair of them. Here we have the constraints about the different types of the nodes. Every conditional edge, which starts from a disjunction node, should be presented with a classic arrow at the end.

Inversely with the open process, method saveGraph writes all the attributes from the model into a CPG file. Meanwhile, if the cell that the method operating on is a node, then it is updated in the hash table nodes, otherwise, it’s updated in the linked list edges. Thus no matter when we

add new cells, or modify the existed cells, the two structures that store the data will get the information synchronously with the cpgModel in order to create and update the CPG file, and maintain the system consistency.

6 Towards an Extensible Visual Editor (EVE)

6.1 Introduction

This chapter presents the design of an extensible visual editor that can be applied easily to many different types of graphs. It has powerful user interface that supports most common operations necessary for the convenient construction and manipulation of graphs.

EVE is extended by using graph type specification files. The specification files describe the elements of the graph type, their properties, and the constraints associated with them. The files are represented using XML.

An overview of EVE is presented in Figure 23.

Specification files are useful either for the user to know how to create a new graph model though the file in text or in XML or other language, or for the application to know how to display a certain type of graph model in its special interface.

The specification is extensible via the formal definition so that other types of graphs may easily have their own specifications. As a simple approach one may attempt to deal with the numerous variations in node and edge decorations via a number of node and edge attributes such as shape, size, color, labels, line style etc. These are mainly what the specifications should take into account.

Besides the specification, the editor should also be able to load graph models from graph files. The same type of graphs shares the same specification but each graph has its own graph file that gives the concrete information how it will be displayed graphically.

The application reads different graph specifications, and the user interface of the EVE will change accordingly, including the tool bars and menus. Since variant graphs have the similarities in the basic structure that constitutes with nodes and edges, the common operations like move, cut, paste, copy, delete, zoom in, zoom out and so on are almost the same, but the command items mostly differ in the operations related with graph cell types. For example in Figure 23, there are “Add a state”, “Add a

transition" and so on in UML statechart editor interface and "Add a normal node", "Add a conditional edge" and so on in CPG editor interface, being presented with different tool bars and menus.

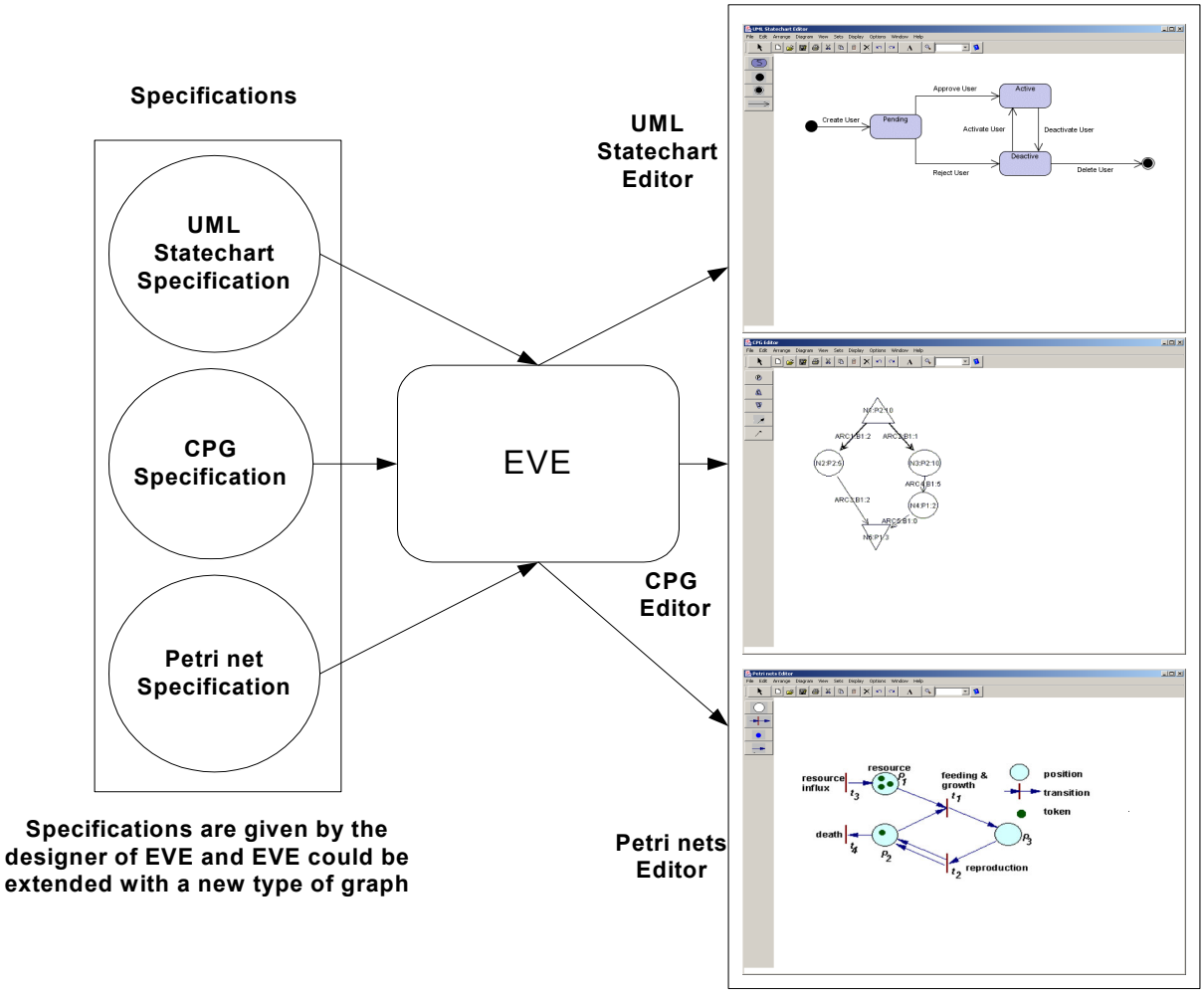


Figure 23. Goal of an EVE

No matter which type of graph being handled, the graph data is always kept in the model and is transferred among the parts of model-view-control, for display and editing.

The design idea is to build general classes for Model, View, and Controller and customize them for each graph type using a graph type specification file:

- Model part handles and stores different node properties and edge properties, so that the viewer and controller could access these data.

- View part gives different visual representations for the nodes and edges based on the data that is required from model part.

- Control part sets the corresponding toolbar and menus for the node and edge types depending on this type of graph specification, handles the model constraints and keeps the communication consistency between the model and view.

6.2 Specifications

Generally, There are mainly three parts in the editor window: menu bar, tool bar and the graph pane. The tool bar consists of a number of command buttons. Some of the buttons are common ones that being provided in many graph editors. These basic buttons include: Select, New, Open (the graphs are loaded in form of graph files), Save (in form of graph files), Print, Cut, Copy, Paste, Delete, Undo, Redo, Text, Zoom, Help and so on. Of course we can add new practical function to improve the toolbar.

In menu bar, we have File, Edit, Arrange, Diagram, View, Sets, Display, Option, Window and Help as the menus for instance. The graph pane is used to display different types of graph being edited.

6.2.1 EVE for CPG

In CPG editor, we simply define three kinds of nodes: normal nodes, conjunction nodes and disjunction nodes, and two kinds of edges: normal edge and conditional edge, and we give three respective command buttons to insert the node, and two buttons to insert the edge, besides the common commands we mentioned above.

Both nodes and edges have text fields which we could enter the short information such as names and resources. We have given more details in the implementation prototype of CPG editor.

Figure 24 gives an example interface of EVE for CPG after we load or create a CPG model.

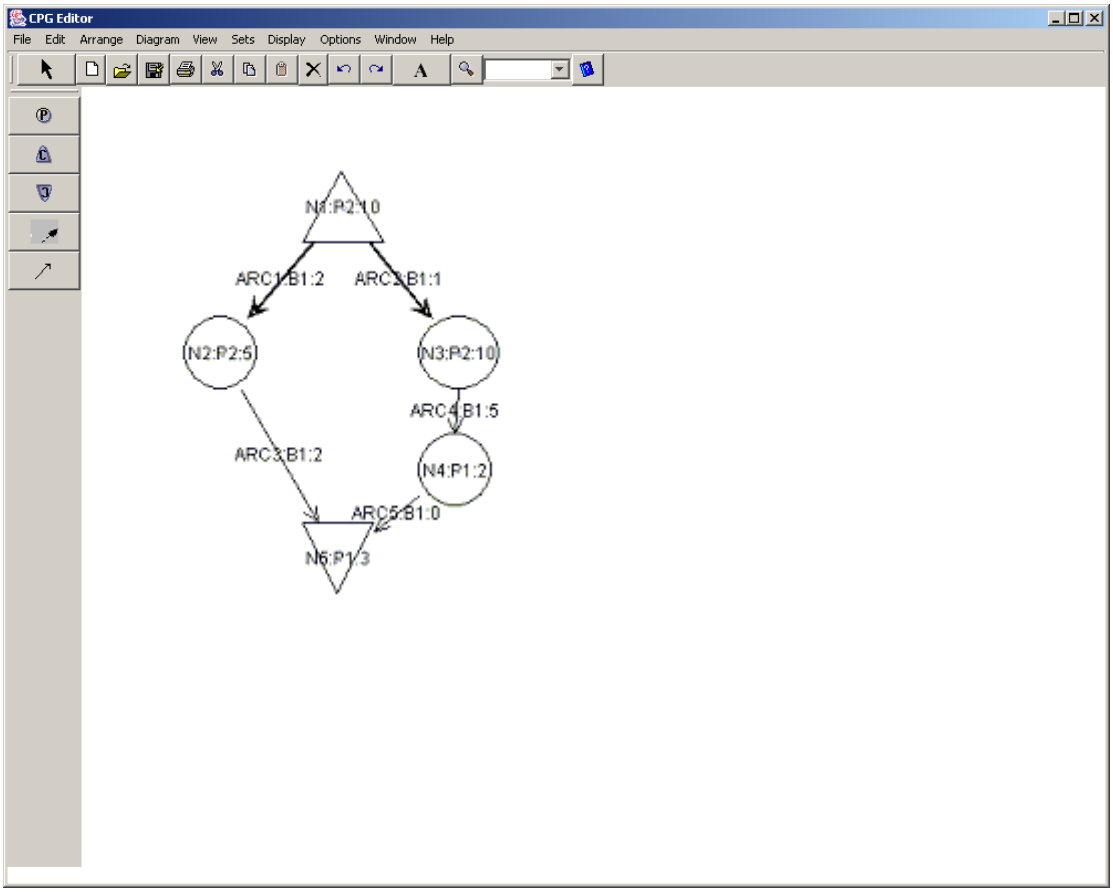


Figure 24. Editor with the CPG

6.2.2 EVE for the Petri net

EVE for petri net may include the features about the selection and manipulation of petri net elements, a toolbar for frequently used operations and the ability to import and export the model as a formal petri net file--which can be user implemented XML file.

The EVE for petri net is similar with the above EVE part for CPG in user interface and main function. We add special operations for petri net such as:

Add a Place

This command is used to add a new place into the panel. Since the place may have the attributes like name, tokens number, value, and description, we could create a small editor dialog instead of a simple text field when double-clicking in the place, which gives more fields to enter and modify these properties.

Add a Transition

Use this command to add a new transition into the petri net. The way to

add is similar with a node.

Add an Arc

This command is used to add a new arc into the petri net connecting the places and the transitions. The way to add is similar with an edge, but the route is from a place to a transition or a transition to a place only.

Add a Token

This command is used to add a new token into a place. The way to add is similar with a node. But it is overlapped on the place.

In Figure 25 we give an example interface after the EVE loads a petri net being edited.

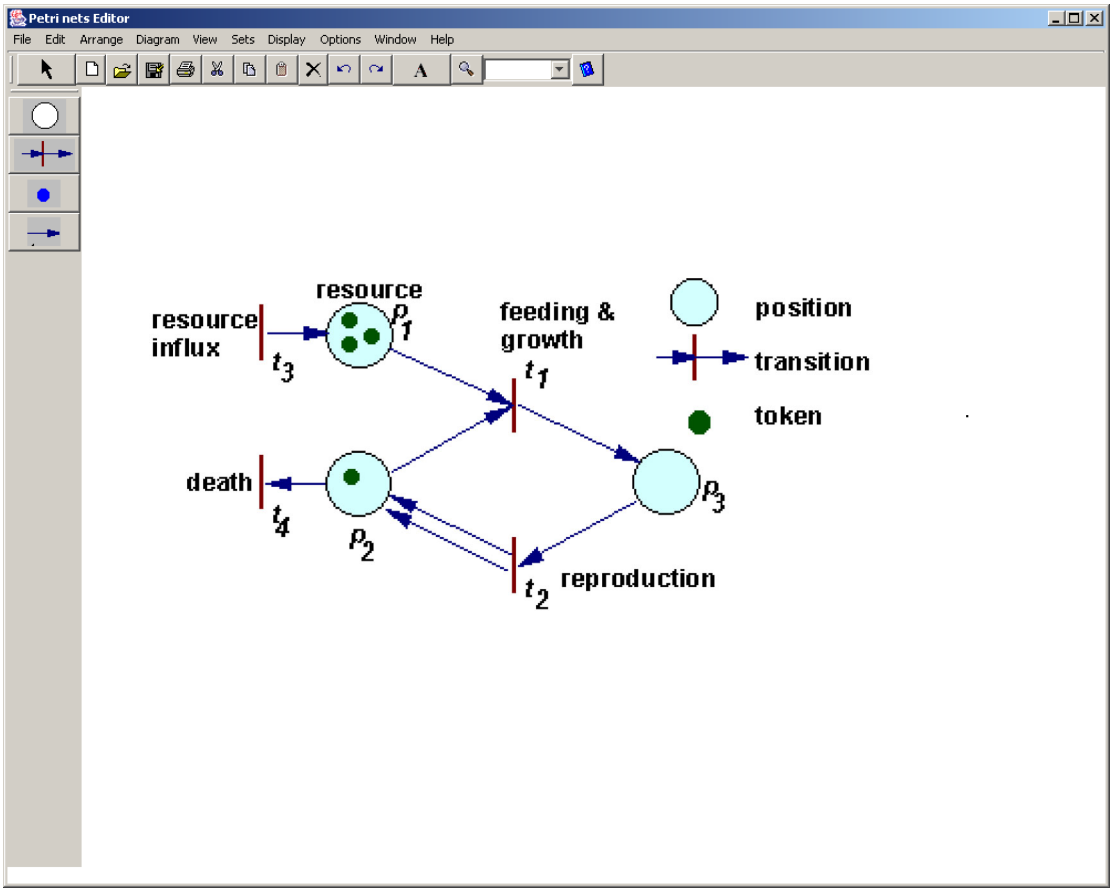


Figure 25. Editor with the petri net

6.2.3 EVE for the UML Statechart

EVE for UML Statechart may include the features about the selection and manipulation of UML Statechart elements, a toolbar for frequently used operations and the ability to import and export the model as a formal statechart file.

The main function and interface of EVE for UML Statechart is still similar with that for petri net. We add special operations for UML Statechart such as:

Add an Initial State

This command is used to add a new initial state into the panel, by clicking in the desired position..

Add a State

This command is used to add a new state into the Statechart. We could get a text field when double-clicking the state, which we could enter and modify the corresponding behaviors.

Add a Final State

This command is used to add a final state into the panel. However, a final state is not always needed in a statechart.

Add a Transition

This command is used to add a new transition into a place. A text field is also needed in order that we could indicate and edit the name of the event trigger. For most statecharts, the transitions are not limited as straight lines, and we could make different designs on it.

In Figure 26 we give an example interface after the EVE loads a UML Statechart being edited.

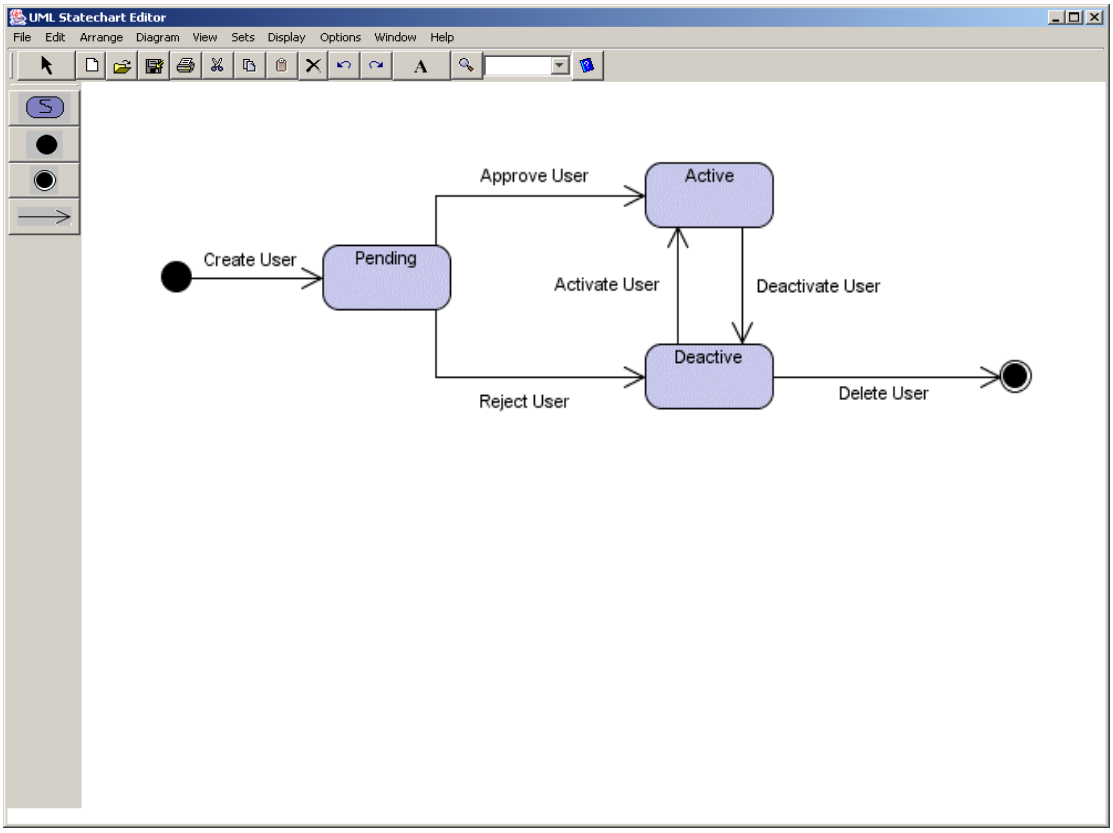


Figure 26. Editor with the UML statechart

6.3 Proposed Software Architecture and Design

In Figure 27, we present the proposed software architecture of the EVE using a UML diagram.

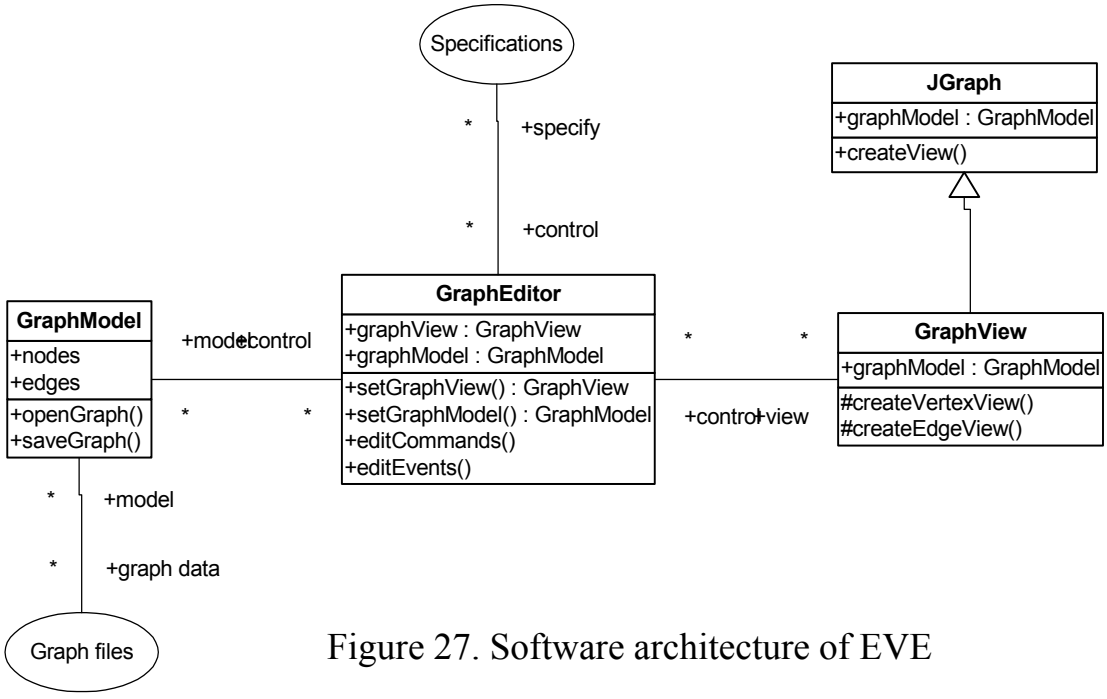


Figure 27. Software architecture of EVE

In this architecture, there are essentially three functional blocks (general classes) that constitute the core of the EVE: GraphModel, GraphView, GraphEditor. They correspond to the model, view, and control, respectively. In the back there are stored graph specifications and graph files that hold the graph definition and data.

GraphModel

GraphModel defines an implementation for suitable data model of the graph and gives the widget access to EVE data. It stores the data and allows the use of any EVE object as a node, port, or edge. This makes it much easier to add visualization to EVE.

The methods defined inside the GraphModel such as openGraph(), provides the access to the graph structure, also the communication between the specifications and the graph models. There might be a lot of boolean expressions like if-else statements inside GraphModel to distinguish the graph type and handle the visualization of different types of graph elements.

Using the design idea from the implementation prototype of CPG editor, we can create several kinds of structures to hold separately nodes and edges objects with respective their properties. And all these properties have a correspondent in the definitions of the graph type specification. The graph files give the concrete data. When we either open from or save into the graph file, the corresponding updates of graph data should always keep coherent with the two structures for each type of graph.

For instance once a CPG graph file is selected, the corresponding CPG is displayed in EVE based on its cells positions and other related attributes that are recorded in the CPG file. After we edit the CPG, the changes that currently hold by the models would be written into the CPG file again, thus GraphModel finishes its transformation between CPG model and the documents including the specification definitions and CPG file examples. For other types of graph, the way that GraphModel works is similar based on the specification.

GraphView

If GraphModel provides the content representation of the graph, GraphView then provides the graphical representation of the graph. It displays the related graph objects using the defined shapes or images in the specification. It allows interactive editing and serves as a simple

front-end to class GraphEditor. The graph gets data by querying its GraphModel object.

GraphView displays its data by drawing individual graph cells. The real drawing is done by each paint method in every cell renderer class. The specification tells the GraphView to create a certain shape or access a certain image for the graph cell. For example, to edit a CPG, when reading a disjunction node from the CPG file being opened, GraphView knows to access a triangle image and draw it on the edit panel because the CPG specification has defined a disjunction node image.

When the application deals with the concrete graph object fetched from the data structures in GraphModel, it creates the corresponding view, and includes its attributes such as names, resources, which are drawn together on the image or shape in the EVE.

GraphEditor

GraphEditor is a general class to create the user interfaces in EVE for different types of graph, and the editor interfaces especially differ in the menu bars and tool bars. The class will define the methods to customize the items in menu bar or toolbar for each different graph type.

The GraphEditor reads the specification first to know the elements definitions for each graph type, for example about the CPG, and then decides to present the tool bar with commands to insert disjunction node, conjunction node, normal node, normal edge and condition edge in the user interface. For other graph types, Then whenever the data that the GraphModel uses or the GraphView draws changes from one graph type to another, the EVE interface would change in the tool bar and menu bar automatically. As an example, Figure 23, Figure 24 and Figure 25 indicates the changes.

GraphEditor could be considered as a controller to mediate and communicate between the GraphView and the GraphModel. GraphEditor is the central class of the EVE. The methods concerning to display or edit the graph, are implemented mainly in GraphView and GraphModel. But both the functions of the GraphModel and GraphView are controlled using the GraphEditor.

6.4 Specifying the Representation Models

EVE is extended using representation model specification files. They have to be written by the designer. The difference between the graph specification and graph file with the same graph type is that although they are both defined in a formal language, a certain graph specification is not allowed to be changed after it's determined until the application needed to be developed, it is a model, a definition for all the graph files with the same graph type, a graph file is just an instance created for a graph model based on the principles set in the specification. Whenever the graph model is modified in the EVE, data in the graph file is certainly updated always.

But within the scope of a certain graph type, the graph specification is primarily used by the application and to present this type of graph visually in a pre-defined uniform way, and to load different edit interfaces.

6.4.1 Specifying Graph Types

The XML specification of the representation model, starts with the graph type:

```
<GraphType>
…
</GraphType>
```

Next, the specification defines the certain respective elements and lists their properties and values for each graph type. Comparison among variant specifications indicates that they might have similar structures but different elements and their attributes. In details, we have nodes, edges needed to specify.

6.4.2 Specifying Graph Elements

Node part in graph specification could be defined like this:

```
<Node type="node_type1" image xlink:href =”directory of the node image”>
  <Property name = “property1” type = property1_type  …(other attributes for
    property1)>
  <Property name = “property2” type =property2_type  …>
  …
  <Property name = “propertyN” type =propertyN_type  … >
</Node>
```

Different graph has different type of nodes, and “node_type1” gives the type name, and the link to the image that is stored in the disk or web

representing for this type of node.

A certain type of graph might have its special and distinct properties for node, and generally, there might be node's name with its data type string, and other attributes--position in center and color in black for example. There might be also node's resource that is a process, with its data type string, the position on the right and color in blue, etc.

The specification should contain all node types for each graph type, and list all their attributes.

Edge part in graph specification could be defined as:

```
<Edge type="edge_type" image xlink:href =”directory of the edge image”
sourceNode = “node_type1”  destinationNode = “node_type2”>
  <Property name = “property1” type =property1_type  …(other attributes for
    property1)>
  <Property name = “property2” type =property2_type  …>
  …
  <Property name = “propertyN” type =propertyN_type  … >
</Edge>
```

It is similar with the definition for node, but one important attribute for edge is to specify the source node and the destination node, in order to create the correct connection.

A certain type of graph has its special properties for edge. Generally, there would be edge's name with its data type string, and maybe position also in center, and edge's resource that is a bus with also string as its data type, etc.

In the next section we give more details about the specifications of each type of graph used in EVE.

6.4.3 XML Specification Files for the Graph Types

Following the principles introduced in the previous sections, we present three XML specification examples for UML statechart, CPG and petri net. Thus to build an EVE, these three specifications could be the models to construct the corresponding graphs. And if there's new type of graph being introduced, the specification could just follow the design rules and be created in a similar way.

UML Statechart

We give a simple example about the representation model created for UML statechart.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!-- Specification for UML statechart -->
< UML statechart >

<Node type="initial state" image xlink:href =”directory of the initial state image”>
</Node>

<Node type="final state" image xlink:href =”directory of the final state image”>
</Node>

<Node type="state" image xlink:href =”directory of the state image”>
   <Property name = “name” type = “string” position = “center” color = “black”>
</Node>

<Edge type = "transition" image xlink:href =”directory of the transition” sourceNode =
   “state”   destinationNode = “state”>
   <Property name = “event” type = “string” position = “center” color = “black”>
</Edge>

<Edge type = "s_transition" image xlink:href =”directory of the transition”
   sourceNode = “initial state”   destinationNode = “state”>
   <Property name = “event” type = “string” position = “center” color = “black”>
</Edge>

<Edge type = "d_transition" image xlink:href =”directory of the transition”
   sourceNode = “state”   destinationNode = “final state”>
   <Property name = “event” type = “string” position = “center” color = “black”>
</Edge>

</UML statechart>
```

In the specification, we specify three types of node and three types of edges based on the definition of a UML statechart. We might have other properties due to the concrete design requirements for a UML statechart editor, and then everything should be put in a similar way.

CPG

Here is a simple example about the representation model created for CPG.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!-- Specification for CPG-->
<CPG>
```

```
<Node type="simple node" image xlink:href =”directory of the node image”>
  <Property name = “name” type = “string” position = “center” color = “black”>
  <Property name = “process” type =”string” position = “right” color = “black”>
  <Property name = “wect” type = “int” position = “right” color = “black”>
  <Property name = “condition” type = “string” value = “null”>
</Node>

<Node type="disjunction node" image xlink:href =”directory of the node image”>
  <Property name = “name” type = “string” position = “center” color = “black”>
  <Property name = “process” type =”string” position = “right” color = “black”>
  <Property name = “wect” type = “int” position = “right” color = “black”>
  <Property name = “condition” type = “string” value = “C”>
</Node>

<Node type="conjunction node" image xlink:href =”directory of the node image”>
  <Property name = “name” type = “string” position = “center” color = “black”>
  <Property name = “process” type =”string” position = “right” color = “black”>
  <Property name = “wect” type = “int” position = “right” color = “black”>
  <Property name = “condition” type = “string” value = “null”>
</Node>

<Edge type = "simple edge" image xlink:href =”directory of the image” sourceNode =
  “simple node, conjunction node”  destinationNode = “simple node, disjunction
  node, conjunction node”>
  <Property name = “name” type = “string” position = “center” color = “black”>
  <Property name = “resource” type =”string” position = “right” color = “black”>
  <Property name = “wect” type = “int” position = “right” color = “black”>
  <Property name = “condition” type =”string” value = “null”>
</Edge>

<Edge type = "conditional edge" image xlink:href =”directory of the image”
  sourceNode = “disjunction node”  destinationNode = “simple node, cunjunction
  node, disjunction node”>
  <Property name = “name” type = “string” position = “center” color = “black”>
  <Property name = “resource” type =”string” position = “right” color = “black”>
  <Property name = “wect” type = “int” position = “right” color = “black”>
  <Property name = “condition” type =”string” type = “integer”>
</Edge>

</CPG>
```

In the specification, we specify three types of nodes and two types of edges based on the CPG definition. Each type of node and edge both has more properties than that in general graph: wcet means the worst execution time of the process the node mapped. For disjunction node, there is a condition and valued it as C, otherwise we give it null value. The resource in node’s property means the process being mapped and in edge’s property means the bus. An edge also has the execution time and condition.

Conditional edges also have a condition associated.

For CPG, a disjunction node only has conditional edges as its output, so we include this constraint inside the specification by listing the allowed source node and destination node for the correct connection when specifying the conditional edge.

Petri net

We give a simple example about the representation model created for petri net.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!-- Specification for Petri net-->
<Petri net>

<Place type = “place” image xlink:href =”directory of the node image”>
   <Property name = “name” type = “string” position = “center” color = “black”>
   <Property name = “process” type =”string” position = “right” color = “black”>
   <Property name = “ect” type = “int” position = “right” color = “black”>
</Place>

<Transition type = “transition” image xlink:href =”directory of the node image” >
<Property name = “name” type = “string” position = “center” color = “black”>
</Transition>

<Arc type="ptotArc" image xlink:href =”directory of the node image”
   source = ”place” destination = “transition”>
   <Property name = “name” type = “string” position = “center” color = “black”>
</Arc>

<Arc type="ttopArc" image xlink:href =”directory of the node image”
   source = ”transition” destination = “place”>
   <Property name = “name” type = “string” position = “center” color = “black”>
</Arc>

<Token type="token" image xlink:href =”directory of the node image” source =
“place” destination = “place”>
<Property name = “name” type = “string” position = “center” color = “blue”>
</Token>
</Petri net>
```

In the specification for petri net, we specify the basic elements: place, transition, arc and token and their relations. The place has the process and execution time as the properties. The arc starts either from a place to a transition or from a transition to a place so we set two types of it. The token is moved between places and this is also specified.

7 Conclusion and Future Work

Embedded systems have are widely used everywhere. Their modeling is performed using several types of representation models. Among representation models, graph-based representations are most commonly used in embedded systems design.

In order to speed up the development of domain-specific visual model editors, generic-modeling environments, which can be extended, have to be developed. In this thesis we propose a generic modeling environment, called Extensible Visual Editor (EVE), which can be extended to handle the class of graph-based embedded systems representation models.

We have done a survey of the representation models including CPG, petri net and UML statechart, and their associated tools, such as JGraphpad for process graph, PIPE for petri net and Statemate for UML statechart.

We have also compared two graph libraries: GEF and JGraph, and concluded that JGraph should be used for the EVE implementation. We have also designed and implemented an editor for CPGs. Using the experience gained from this implementation, we have proposed several design ideas for EVE.

The main idea was to extend EVE using representation model specification files written using XML. We have developed three such specification files, for each representation model considered.

As the future work, we need to finalize the design of EVE, and provide a prototype implementation.

We should also complete and improve the specifications for different types of graph depending on the improved design for the extensible editor.

References

1. Andreas Gerstlauer, Daniel D. Gajski “System-Level Abstraction Semantics”. 2002.

2. Dia, http://www.lysator.liu.se/~alla/dia/

3. Douglass, Bruce Power “UML Statecharts”, http://www-md.e-technik.uni-rostock.de/ma/gol/ilogix/umlsct.pdf

4. GME 2000, the generic modelling environment, http://www.isis.vanderbilt.edu/Projects/gme/default.html

5. GEF, http://gef.tigris.org/

6. GEF,IBM, www.cs.technion.ac.il/~cs234307/2002-2003/GEF_YPP.pdf

7. K. Jensen “Coloured Petri nets. Basic Concepts, Analysis Methods” (Vol. 1). Ed. Springer-Verlag.

8. GXL, http://www.gupro.de/GXL/

9. GTK+, http://www.gtk.org/

10. J2SE, http://java.sun.com/j2se/

11. JGraph, http://jgraph.sourceforge.net/

12. Luis Alejandro Cortés, Petru Eles, Zebo Peng “A Survey on Hardware/Software Codesign Representation Models”. 1999.6.

13. Meta Edit+, http://www.metacase.com/methods/

14. Microsoft Visio 2003, http://www.microsoft.com/office/preview/visio/overview.asp

15. Paul Pop “Scheduling and Communication Synthesis for Distributed Real-Time Systems” .2000.

16. Petru Eles, Krzysztof Kuchcinski, Zebo Peng, “Scheduling of

Condtional Process Graphs for the Synthesis of Embedded Systems".

17. PGML, http://www.w3.org/TR/1998/NOTE-PGML-19980410

18. PIPE (Platform Independent Petri net Editor), http://petri-net.sourceforge.net/

19. Predator Editor, http://cadgraphicswest.com/html/predator_editor.html

20. Robin.J.Wilson "Introduction to Graph Theory". Ed. Academic Press. INC.

21. Statemate, http://www.ilogix.com/products/magnum/index.cfm

22. SVG, http://www.w3.org/TR/SVG/

23. Syed Zia Akbar Zaidi "Portable Automotive Electronic Models Using Standard XML Technologies". 2002.

24. Wayne Wolf. "Computers as Components, Principles of Embedded Computing System Design". 2001. http://www.ee.princeton.edu/~wolf/embedded-book/overheads/

25. Wayne H. Wolf "Hardware-Software Co-Design of Embedded Systems". 1994.

26. W. Reisig "A Primer in Petri net Design". Ed. Springer-Verlag.

27. UML Statechart, http://www.dotnetcoders.com/web/learning/uml/diagrams/statechart.aspx