Graphical Editor for the Metropolis Meta-model

Jiagen Ding (jgding@newton.berkeley.edu)
Hongjing Zou (zouh@eecs.berkeley.edu)
http://www-cad.eecs.berkeley.edu/~zouh/ee249project.html

Abstract

The goal of this project is to provide a graphical editor to navigate the structural hierarchy of a design developed using the Metropolis meta-model, inspect the code and constraints associated with processes and communication channels, and generate the meta-model code to instantiate the objects in the hierarchy using the meta-model syntax.

1 Introduction

Reusability is a major concern in modern-date system designs. To increase reusability, functions of a system are designed as independent as possible [7, 2, 4]. Two important ideas are introduced to achieve this. One is to keep the system function and the system architecture separated; the other is to keep computation of functions and communication of functions separated [3]. Hence in nowadays designs, there are levels of abstractions and different models of computation (MOC) [5] are used for different levels of abstraction. To effectively manage such complex design tasks required in modern-date designs, establishing formal design methodologies is imperative. The Metropolis [6] is developed to build an environment for modern system designs, in which systems can be unambiguously represented throughout the abstraction levels, the design problems can be mathematically formulated, and tools can be incorporated to solve some of the problems automatically.

For such a design environment, a friendly graphical user interface (GUI) is needed for the ease of using the environment, and this comes the motivation of our project. We should first study and get familiar with the design methodology of the environment, i.e., in what way would a designer design his system using the Metropolis. Only after that could we know the requirement of the graphical editor and design its functions.

The implementation of the graphical editor is by adapting the Diva graph package [1], to realize the functions that we needed.

The report is organized as follows. In section 2, a brief review of the Metropolis meta-model is given, where both the methodology and the infrastructure of the meta-model are reviewed. In section 3 the graphical editor is introduced. The structure and the implementation are described in particular. Section 4 concludes the project and gives our ideas of improvement and future work of the graphical editor.

2 Metropolis meta-model

The Metropolis is a design environment that consists of an infrastructure, a tool set, and design methodologies for various application domains. The infrastructure provides a mechanism such that (1) heterogeneous com-
ponents of a system can be represented uniformly, and (2) tools for formal methods can be applied naturally. Here we are going to review the design methodology and the infrastructure.

2.1 Design methodology

The design methodology proceeds from the highest level of abstraction to the implementation through a series of refinements. The goal is to increase the reusability of components that define the system functions. This is achieved by orthogonalizing issues concerned in each design step in such a way that independence of decisions taken in the steps can be maintained as much as possible. Designs using this methodology follow the fundamental steps summarized below.

Functional decomposition

At the highest level of abstraction a system is considered as a single process that describes the desired function. This process is decomposed into a set of concurrent processes, each of which establishes a relation between input and output domains.

Behavior adaptation

When two functions defined above are connected, a behavioral adapter is inserted between them, to transform the domains of the functions so the outputs from one function can be taken as input to the other. By explicitly inserting adapters, one can maintain the reusability of the system components defined in the first step. The adapters are modeled as processes.

Communication

Each of the links connecting two processes is assigned a particular communication semantics, which is specified by defining a medium in the meta model. Thus communication is separated from computation. At the same time, each process, if needed, is wrapped by another special process (scheduler) that specifies firing conditions of the original process and its interaction with the connected media. In this way, changing the media requires changing only the wrapper (scheduler), without affecting the main process.

Communication refinement

Each medium is further replaced by a network of processes and media. This refines an abstract communication mechanism into an interaction of components that are more easily implemented.

Mapping and Optimization

The refined system is mapped onto an architecture of components. Noticed that the function and architecture is kept separated. So that whenever redesign is needed, one can always return to models in higher level of abstraction where modification can be kept local without affecting the system functions. After the mapping, an optimization is carried out. While the previous steps are concerned about reusability, they may introduce redundancy. To get better implementation, the result after mapping is further refined and optimized.

2.2 The infrastructure

The core of the infrastructure is a meta model of computation, which allows one to model various communication and computation semantics in a uniform way. Here we are mainly concerned about the way in which the meta-model presents the design, because this is the way in which the graphical editor would show to the user.
From the review in last section, we know that there are processes representing system functions. Sequential processes are concurrently communicating each other through the media. A medium is defined with a set of variables and functions. Values of the variables constitute states of the medium. The variables can be referenced only by the functions defined in the medium, and communication among processes is established by calling these functions to change or evaluate the states of media. This mechanism allows communication semantics to be modeled solely by specifying media, independent from process specifications. In this view, a process is an active object in the sense that it specifies what it computes and what it wants to communicate by interacting with media, while a medium is passive as it only provides a means for processes to communicate.

Scheduler is a special type of processes. It is used to specify how to coordinate execution of concurrent processes. The scheduler is statically connected to processes to be scheduled. It observes states of processes, and controls the execution.

Process, medium, and scheduler are represented by nodes in the graphical representation and they are connected by edges. Figure 1 shows the network of the infrastructure.

3 The graphical editor

The basic meta-model is represented by the network of the three kinds of nodes, i.e., process, medium, and scheduler, so the basic function that the editor should provide is to let the user manipulate the nodes, i.e., create, modify, and delete them by key board input or mouse click. The editor should also allow the user to connect the nodes in the way they have been designed and generate the netlist of the network for other supported tool like simulation. Furthermore, it should provide subgraphs for refinement.

In requirement point of view, the editor should be dynamic and interactive. Dynamic property requires the editor display immediate and relevant feedbacks about properties of the systems. Interactive property requires that the editor provide a friendly interface and give hints.

3.1 The structure

The graphical editor we provide for this project is an adaptation of the Diva software infrastructure. The core structure of the editor is an MVC data and notification model. Figure 2 shows the structure of the graphical editor.

The MVC model consists of a Model, a View, and a Controller interact with each other. The Model is a database in which the information of nodes and edges are stored. The View accepts the user’s command and displays information of the system. The Con-
Figure 2: The structure of the graphical editor. It is designed as a MVC (Model, View and Controller) data and notification model. The Controller is the core for the editor.

troller is the most important part in the structure. It recognizes input and decides what to do. Besides the three parts described above, there is another component called Renderer in the structure to create figures. It defines how the figures look like, i.e., circle or rectangular, etc.

### 3.2 Implementation

We implement the meta graph editor based on Java swing package and diva package. The core classes for the meta graph editor is MetaGraphModel, MetaGraphController, and MetaGraphEditor.

MetaGraphModel is the graph model which serves as a database for storing the meta model in terms of nodes and edges. Each Node is a semantic node, representing a computation process, a communication medium or a scheduler. We call these semantic nodes as actors which are implemented as ProcessActor, MediaActor and Scheduler Actor. Each Actor has its name, type, and associated codes.

```java
public class ProcessActor extends Object{
    private String _type;
    private String _actorName;
    private String _portName;
    private String _codeText;
    //constructor
    public ProcessActor(){
        super();
    }

    ... public String toString(){
    return _actorName+_type
    +_portName+_codeText;
    }
}
```

MetaGraphController is the graph controller which deals with graph events. MetaGraphController includes a node controller and an edge controller. A node in the graph model may be composite, consisting of an entity and several ports. Thus, the node controller consists of an entity controller and a port controller, which controls the behavior of the entity and ports respectively.

MetaGraphEditor is the class includes the “main” function. This class is essentially a MDIApplication, which provides a screen display for building, modifying, inspecting and saving the meta graph model. This editor provides menus and toolbars, a tree panel and an editable meta graph panel. The menus and toolbars provide functions such as new, open, save, save as, exit and netlist generation. The tree panel displays the nodes, which user can drag drop them to the meta graph editor panel which contains the canvas component “JGraph”.

A MetaDialog class is provided for inspecting the node codes and constraints. When the user presses the “P” key, a MetaDialog will pop-up up, which displays the semantics of the selected node. In this dialog, user can change the node name, add/delete ports, specify the port position, and modify the codes for this node. Notice only compos-
ite nodes are allowed to add ports to them.

To realize the function of netlist generation, ports are defined for the process nodes and the scheduler nodes. Netlist Generator is the class responsible for netlist generation of a given meta graph model. Based on the graph model, the netlist generator will create the node instance and add it to the netlist. The netlist generator also connect the nodes via appropriate ports.

```java
for each process node and scheduler node {
    for each port {
        connect(SrcObject, SrcPortName, DestObject);
    }
}
```

ProcessRenderer, MediumRenderer, and SchedulerRenderer are key classes in the for the Renderer. Given a type of node, they return a rectangular, a circle, or a diamond for process, medium, or scheduler node respectively.

### 3.3 How it works

The graphical editor works in an interactive way between the Viewer, the Model database, and the Controller. The Renderer always creates figures for the Controller and Controller decides what to display.

We use an example of modifying the code of a node to show how the graphical editor works. First, the user will click on the node that he is going to modify. The Viewer receives the mouse click and sends it to the Controller. The Controller gets the input, analyzes it, then sends the Viewer command to display that the particular node is being selected. Second, the user press P on his keyboard for property of the node. Again, the Viewer receives it, and sends it to the Controller. When the Controller gets this, it sends inquiry to the Model database, and the Model returns the property of the selected node to the Controller. Then, the Controller sends command again to the Viewer to display a dialog on screen to show the node. The user will then modify the code in the dialog. When he finishes, he will send command save to the Viewer, then to the Controller. At this step, the Controller would understand the command save and send the new code to the Model database. The Model updates the property of the node. Now, the modification is done. Next time when the user opens this design, the modified code will be displayed as well as be used in any other tools such as simulator.

### 4 Conclusion and future work

The project gives us a better understanding of modern design methodologies as well as training on Java programming. We reviewed the essential idea of modern embedded system design: orthogonalization of concerns. We learned about the infrastructure of an environment provided for such designs. And we practiced our programming skills. As inexperienced Java programmers, we have finished the basic functions of the graphical editor. Some bugs need to be fixed such as checking of illegal connections, i.e., connections between scheduler node and process node. Refinement needs to be added for netlist generation.

The graphical editor of this project is a one-way editor, i.e., given the graphics, output the netlist. Future work can include the reverse of it, i.e., given the netlist, output the graphics. Also, new package of graph model database should be provided to fully exploit functions provided by Metropolis design environment.
Acknowledgment

We would like to thank our mentors Yosinori Watanabe and Luciano Lavagno. And we would also like to thank Stephen Neundorffer for help on the Diva package.

References


