The Building of A Micro Network Modeling Toolkit by Marama

Chunhui Han

Mcgill University, Montreal, Canada

Abstract. Marama is a set of tools for building Domain Specific Software Tool in Eclipse. This paper describe the building of a network modeling toolkit in Marama, which can be used to construct simple network topology and to simulate basic network events.

1 Introduction

Domain-specific visual language tools have become necessary in many domains of software engineering and school course teaching. Especially for the domain of Network Modeling, many powerful network model simulator softwares have been developped, such as ns-2, Shunra and Boson. Both of these toolkits have some common features: they provide an overlay experimentation platform for creating virtual micro Internets for teaching and learning; they simulate some network events and give feedbacks to the user-input commands; The parameter obtained from the simulation are significant and can be applied to the real network.

However, building such tools from the scratch is very challenging because of the long developing cycle and the difficulty of satisfying multiple user requirements. Therefore, it would be very interesting to find a developing mode which supports fast design and user-defined function.

Marama[1] is a meta-tool which aims to support users to rapidly design the prototype and to evolve tools to support a wide range of domains. Marama is developed by University of Auckland, based on their previous meta-model project, **Pounamu**[2]. Marama is developed as an Eclipse plug-in and a set of Marama Meta Tools are being implemented to replace the Pounamu tool specifications, even though Pounamu specified tools are still loadable and functioning in the Marama environment. Now Marama integrates the features of both meta-tools and modeling tools. Marama uses Marama Metamodel Definerview to specify tool metamodel, Marama Shape Designer view to specify tool shapes and connectors, Marama ViewType Definer view to specify mappings of meta-elements to visual representations. It also supports complex behaviour specification via OCL constraints, visual event handlers and a comprehensive API. Marama uses Marama Diagram view to create model instances or independent views of selected view type.

2 Approach

Marama performs excellently in the desing of Meta-Model and Shape-Model, however, its Behaviour design part doesn't provide users much freedom to define model events according to their own willings. Therefore, we decide to bring in some Java programs to handle the model-event part. Basically, our approach consists of two aspects: building the Network Modelling Toolkit using Marama meta-tool, and simulating the network behaviours using the external java program.

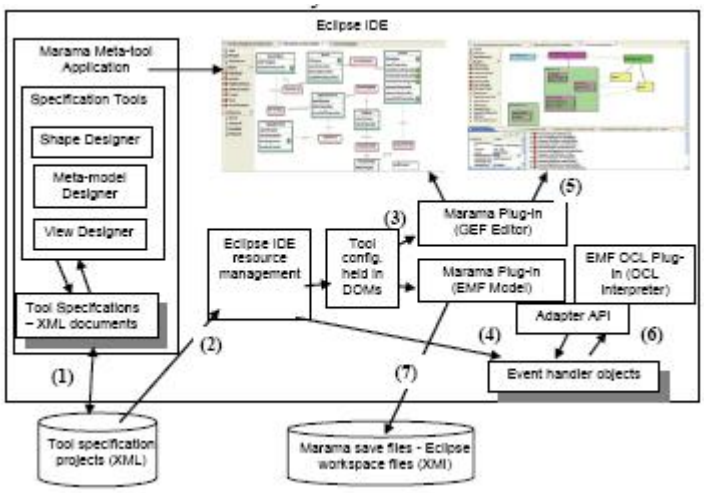

Fig. 1. Marama Structure

2.1 Marama Part

A set of Marama meta-tool are implemented to build Domain-Specific Model Language, including Marama Meta-Model Definer, Marama Shape Designer and Marama ViewType Definer.

We use Marama Meta-Model Definer to specify the underlying meta-model of the Network Toolkit, which contains the network component entities with attributes, and the relationship between them. See Fig.2.

An entity type can be created by dragging and dropping an EntityShape from the editor palette. We name the entity-type by changing the name property of the EntityShape in the Properties view. An association type can be created by dragging and dropping an AssociationShape from the editor palette. We name the association type by changing the name property of the AssociationShape in the Properties view. Then, we must specify the Association end types and association end multiplicities. As seen in the Fig.3, we change the value of the property of the **Workstation** EntityShape in the Properties view.

We use Marama Shape Designer to define the icon shapes for each meta-model entity and relationship. See Fig.4.

The ShapeShape tool from the palette is used to create owing (base/container) shapes. Each owning shape instance is provided with a shape viewer to visualise the shape design. A name property must be assigned for a ShapeShape as its identifier.

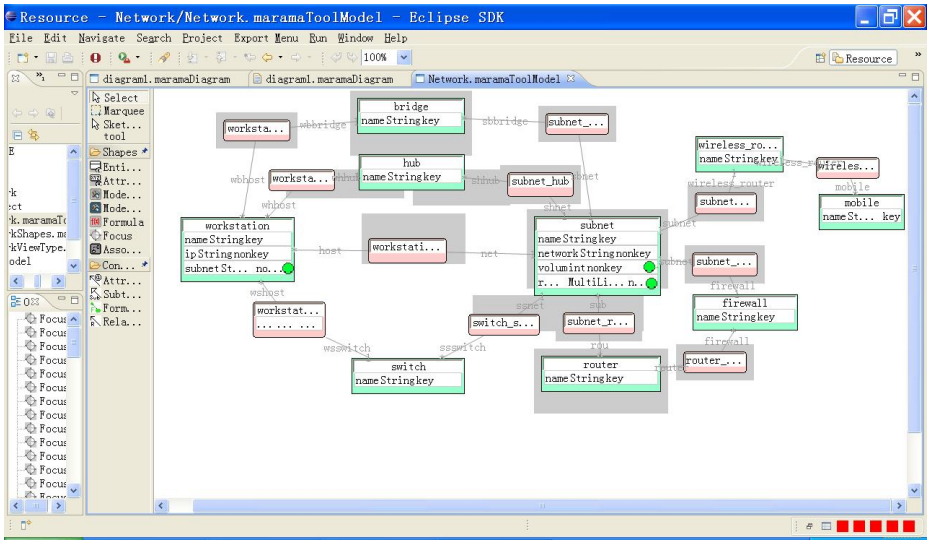

Fig. 2. Meta Model View

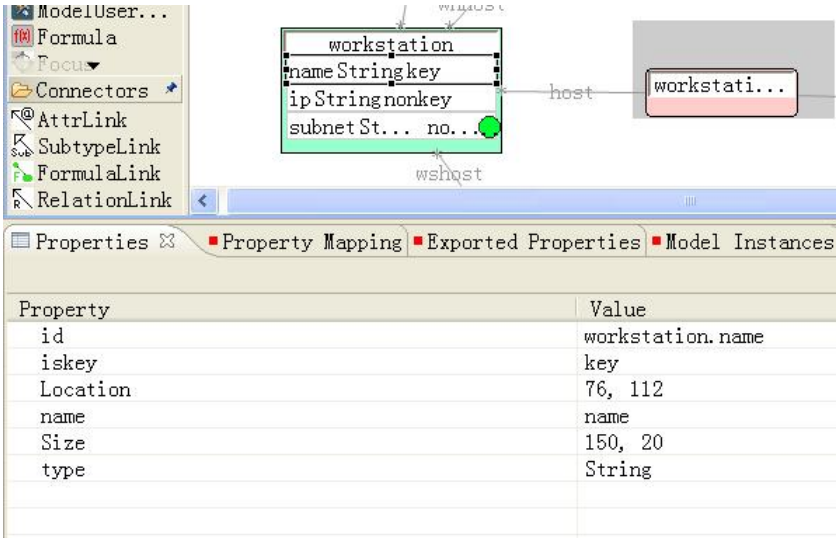

Fig. 3. Change Entity Properties

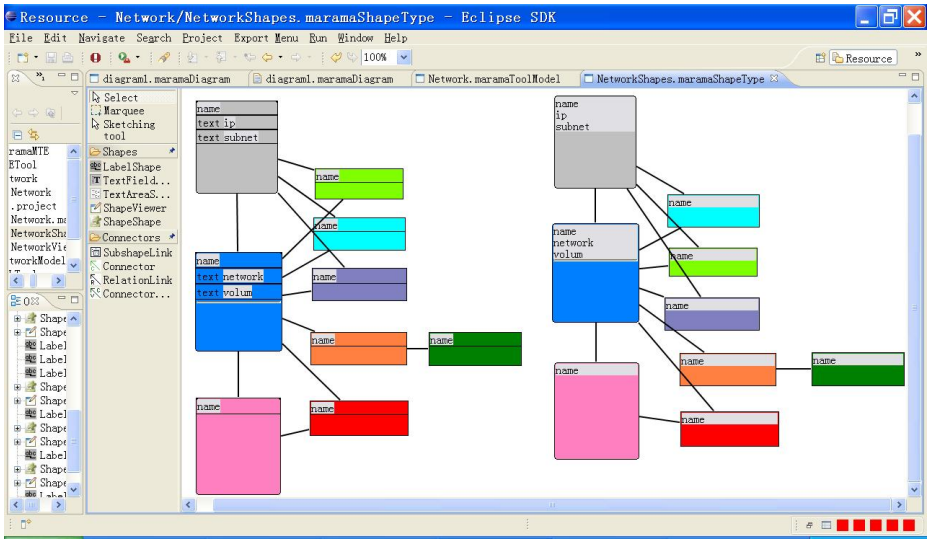

Fig. 4. Shape Model View

The LabelShape, TextFieldShape, TextAreaShape and the ShapeShape tool from the palette are used to create sub shapes of an owning shape. A sub-shape is automatically assigned with a unique identifier (the name property) when it is added to an owning shape with the specified container layout constraint (e.g. null layout, flow layout, vertical layout, border layout and grid layout). A ShapeShape can be created as a subshape as well as an owning shape.

The Connector tool in the palette is used to design connectors. Each connector instance is provided with a connector viewer to visualise the connector design. A name property must be assigned for a Connector instance as its identifier.

number of shape and connector properties can be exported. Exported properties are those that can be mapped to metamodel properties and set at runtime in instances of a shape or connector by the user. As seen in the Fig.5, we export the property **name** of the **Workstation** shape.

We use Marama View Designer to specify mappings of meta-model entities to visual representations. As shown in Fig.6, the ViewShape tool from the palette is used to add shapes to the view type and the ViewConnector tool from the palette is used to add connectors to the view type. Available shapes and connectors for the tool project are automatically seen and selectable to map to a view shape or view connector. The

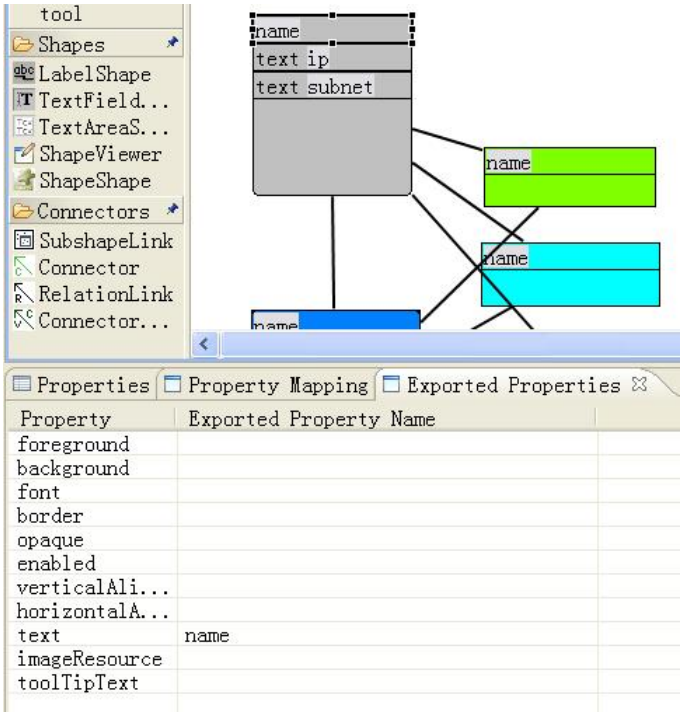

Fig. 5. Export Shape Property

user could set the name property of a view shape or view connector by selecting it from a dropdown list in the properties view.

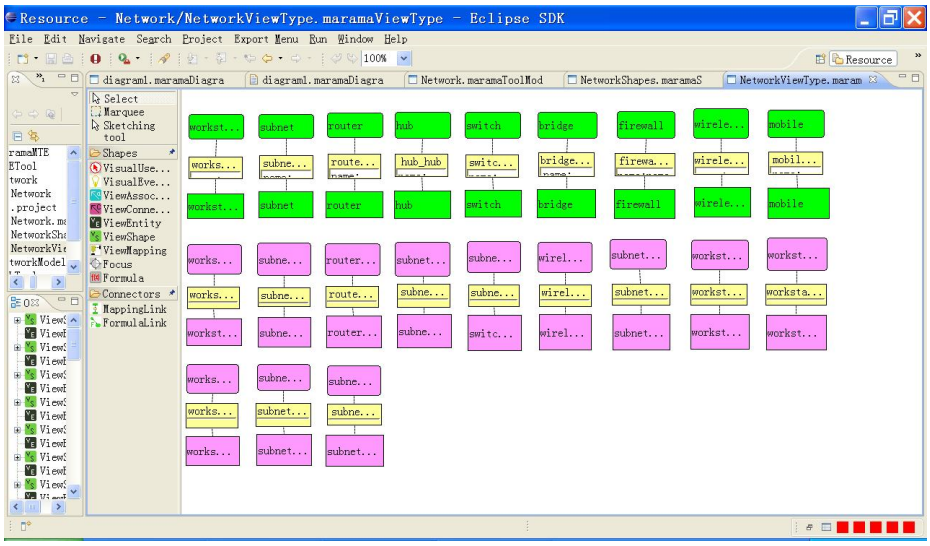

Fig. 6. Mapping of Meta Model to Shape Model

Property mappings between a view shape and a view entity or between a view connector and a view association are specified in the Property Mapping view. As shown in Fig.7, we map the Entity property **name** of **Workstation** to the Shape property **name** of **Workstation**.

Marama provide some behaviour specification features, such as Fomula, which can be added to the meta-model view to achieve the value-dependent runtime behaviours. For instance, as shown in Fig.8, we add a fomula which makes the property **subnet** of Workstation shpae be set to the property **name** of the Subnet shape it connect with.

After the tool definition is finished, user can create a model project based on a selected modeling tool definition using the Marama Model Project wizard, and build a network model instance in the Marama Diagram view. See Fig.9.

The instance model information is stored in a XML file, which record the properties of each meta-model entity and their relationship. This XML file makes it possible for us to use external Java program to access the Marama Instance Models.

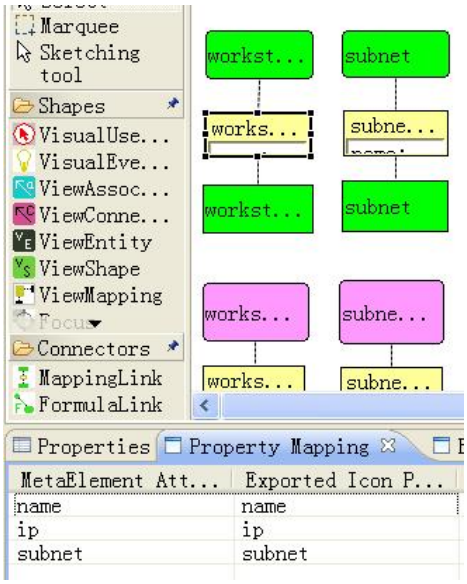

Fig. 7. Map Model Properties

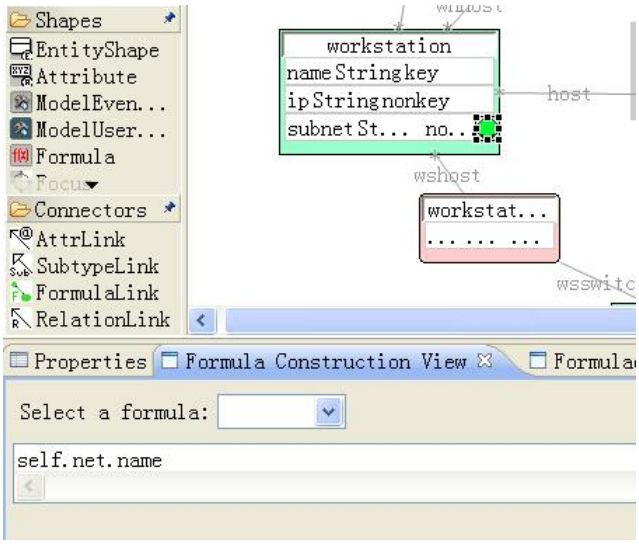

Fig. 8. Add Fomula Constrains

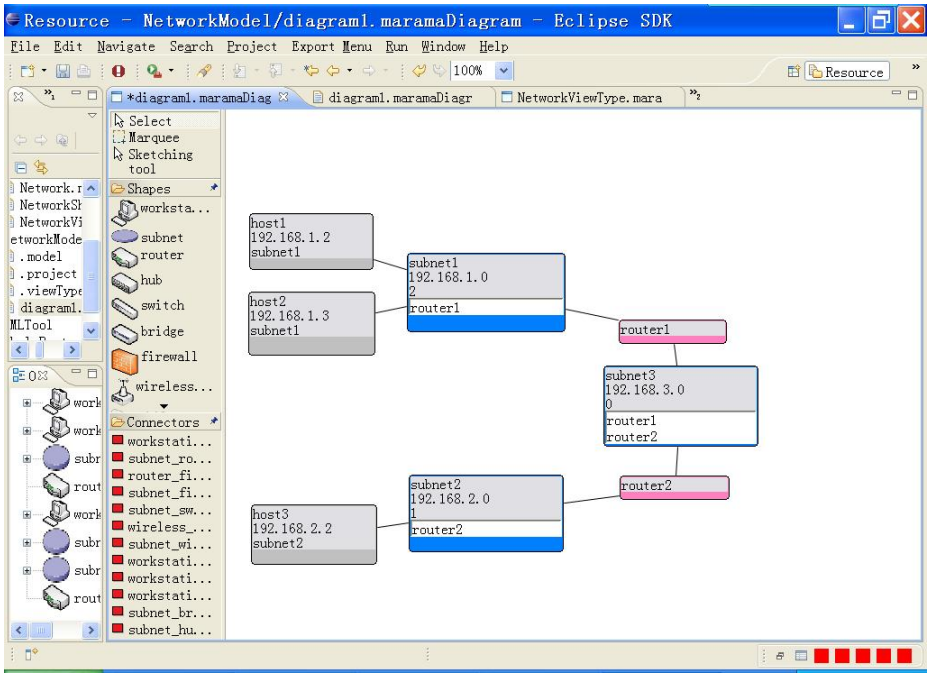

Fig. 9. An Network Model Instance

2.2 Java Part

An external Java program is implemented to simulate the network behaviors based on the network model. The external Java program contains two main modules, **Center Module** and **Client Module**.

Center Module The Center Module uses JAXP (Java API for XML Processing) to process the model XML file generated from the Marama tool. Then, the network topology is built according to the information extracted from the Instance Model XML file. The Java Class structure of Center Module is shown below:

- **CenterModule.class**: read the XML file and process it; start a center thread to listen for the commands from the Client Module and give response.
- **NetworkFuntion.class**: hold the Router, Subnet and Host objects of the network, which define the whole network topology; provide some very important methods for CenterModule to call.
- **Router.class**: define an object to represent the Router component in the model.

– **Workstation.class**: define an object to represent the Workstation component in the model.
– **Subnet.class**: define an object to represent the Subnet component in the model.
– **RouteEntry.class**: define an object to represent an router entry, which is stored in the Router object.

The Center Module try to simulate the real router working scheme. For instance, it looks up the routing table of each router to forward the packets among hosts in the network model. However, the Java program only simulate the behaviours in the **Network Level** and ignore the **MAC Level**.

Client Module We developed two sort of Client Module: Host Module and Router Module. The Client Module provides a virtual terminal interface to simulate the real Router and Host terminals. The user-input commands are sent to the Center Module, and processing result are sent back to display on the terminal. The basic structure is shown in Fig.10.

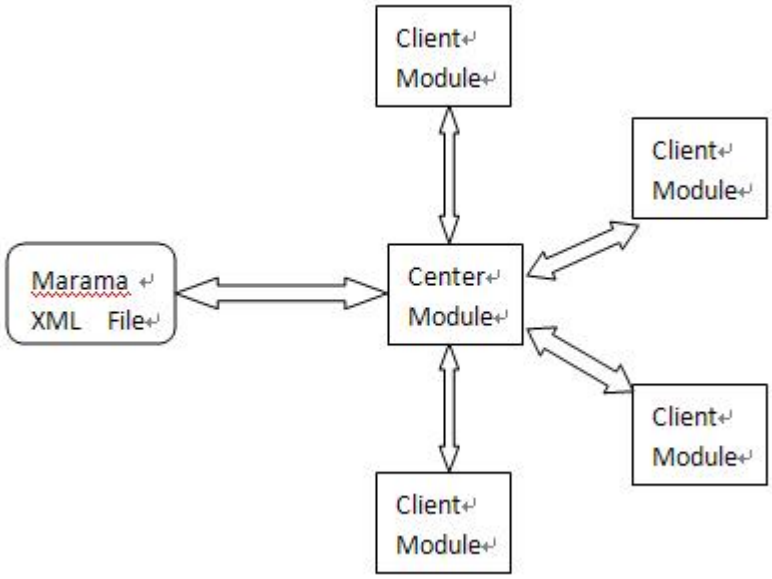

Fig. 10. Java Modules Structure

3 Test

To test the correctness of network topology generated by the Java program, we developed a variety of functions to simulate the network commands. The testing result is shown in Fig.11-Fig.18.

First, as seen in Fig.11, we start the CenterModul to process the XML file. The input parameter is **diagram1.xml**, which is the XML file generated by the network model in Fig.11.

Fig. 11. Center Module Terminal

We start a Host ClientModule in Fig.12, by specifying the IP address of the host workstation.

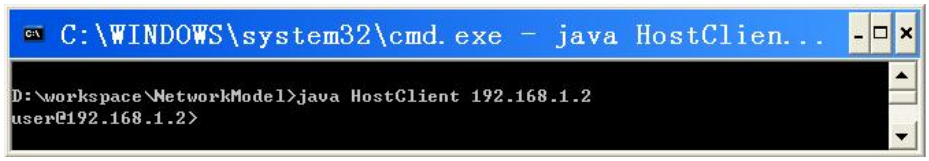

Fig. 12. Start Host1 Module

Now, input **route** and you can see the routing table for this host. As the network topology is generated, the routing table for each host is also automatically generated. As show in Fig.13, the default route for each destination is **192.168.1.1**, which is the IP address of the first interface of **router1**.

And then we can start another Host ClientModule, **192.168.2.2**, and do the same work.

Now, in the terminal of host **192.168.1.2**, try to input **ping 192.168.2.2** to test the the connection between these two machines. As shown in Fig.14, you will find the **192.168.2.2** fails to give you response.

This is because the routers doesn't know where to forward the **ping** packet. We need to physically input some static routing entries in the two routers, to tell the routers the how to work. As shown in Fig.15, we first

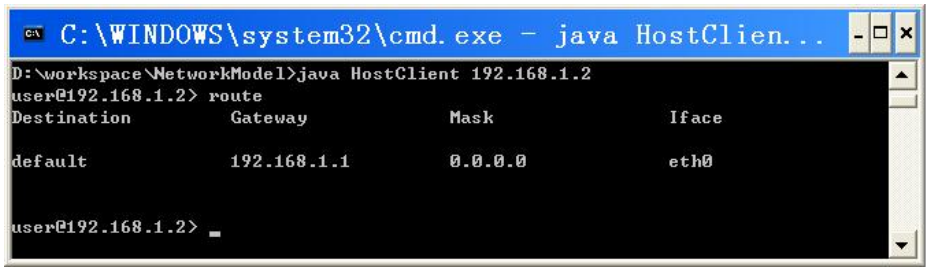

Fig. 13. Host1 Route Table

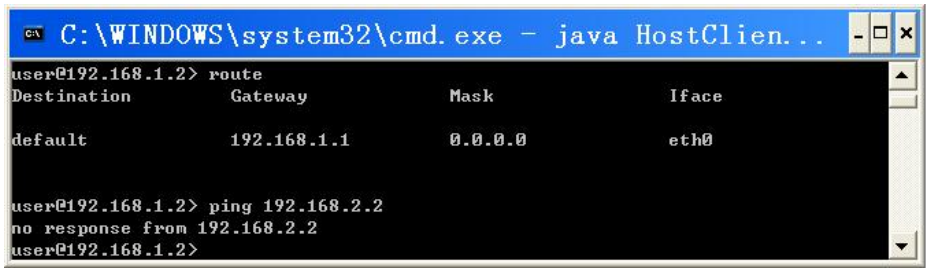

Fig. 14. Host1 Fail to Ping

start the Router ClientModule by specifying the router name **router1**, and similarly for **router2**. Then, in Fig.16 and Fig.17, we use **route add** command to add two route entries for both **router1** and **router2**.

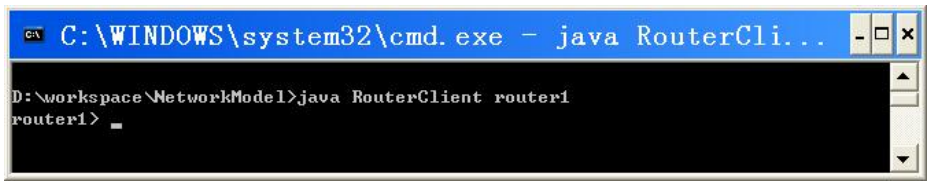

Fig. 15. Start Router1 Module

After that, you can input **route show** to check if the route entries are correctly added. See Fig.18.

Now we try to **ping** between **192.168.1.2** and **192.168.2.2**, and they receive reponse from each other. See Fig.19 and Fig.20.

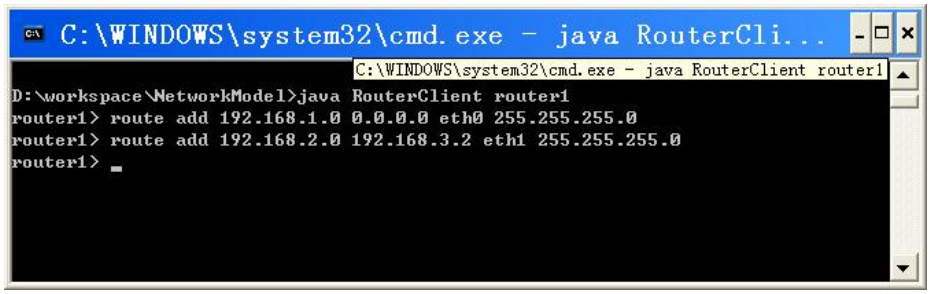

Fig. 16. Add Route Entries for Router1

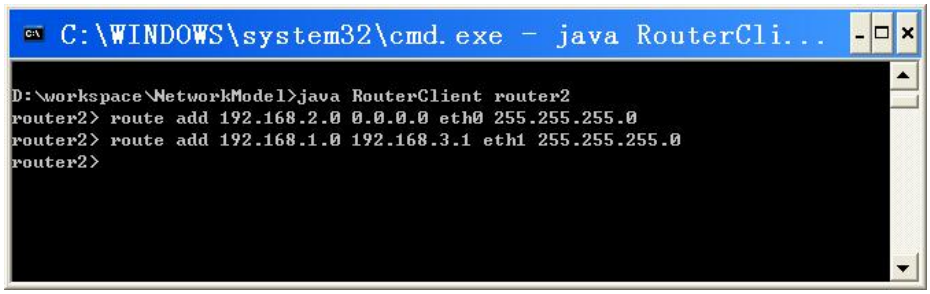

Fig. 17. Add Route Entries for Router2

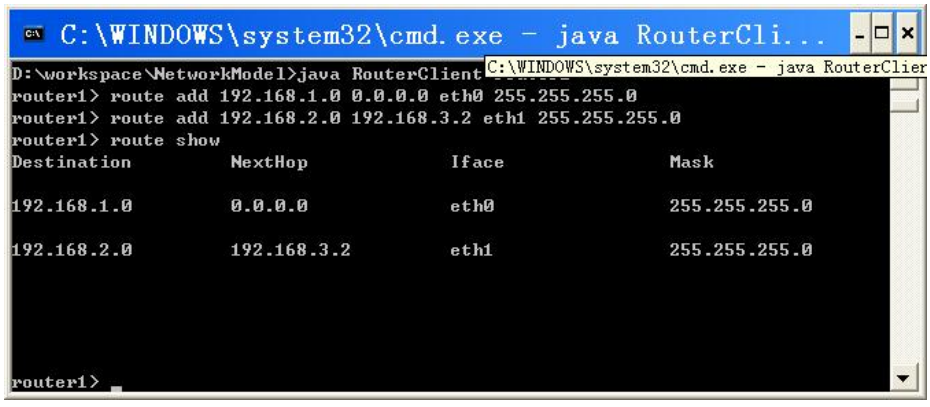

Fig. 18. Show Router1 Route Entries

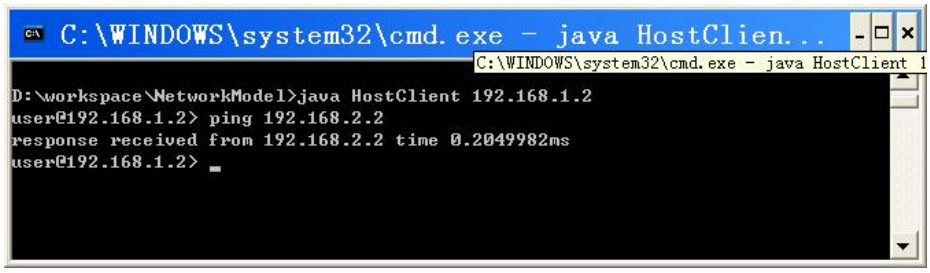

Fig. 19. Host1 Ping Host2

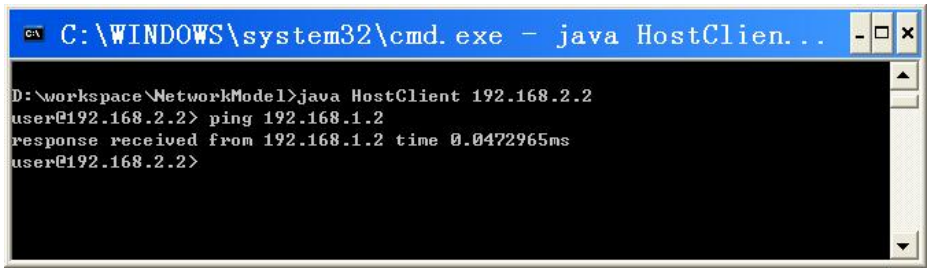

Fig. 20. Host2 Ping Host1

4 Summary

We used an Eclipse-based meta-tool, Marama, to develop a network domain modeling toolkit. In addition to this toolkit, an external Java program is implemented to simulate the network behaviours. In the current toolkit, user can add static route entries and ping each host in the network. In the future work, we will add some new features and simulate other complex network commands.

References

1. John Grundy, John Hosking, Jun Huh, Karen Li.: Marama: an Eclipse meta-toolset for generating multi-view environments 2007 Workshop on Software Innovation and Engineering New Zealand. **Nov** (2007) 26-27
2. Nianping Zhu, John Grundy, John Hosking, Na Liu, Shuping Cao and Akhil Mehra.: Pounamu: a meta-tool for exploratory domain-specific visual language tool development. Journal of Systems and Software. **80** (2007) 1390-1407