Version 4.5

The Call Processing Language Example

MetaCase Document No. CE-4.5

First Printing, 2nd Edition, February 2008.

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä
Finland

Tel: +358 14 4451 400
Fax: +358 14 4451 405
E-mail: info@metacase.com
WWW: http://www.metacase.com

You may order additional copies of this manual by contacting MetaCase or your sales representative.

Preface

The goal of this example is to demonstrate Call Processing Language (CPL) in MetaEdit+. The call processing framework and language (Lennox et al 2004) presents an architecture to specify and control Internet telephony services. The CPL language in MetaEdit+ allows service engineers to specify call processing services by using directly the CPL concepts.

This example will cover the issues of CPL language use and code generation. First we inspect the CPL language and its key concepts and after that we use it in modeling. Finally we generate call processing service definitions in XML format. Please note that certain parts of the example may require working hands-on to ensure the best understanding of the subject matter.

For exploring the CPL example, the following things are required:

- MetaEdit+ for trying out the CPL language. The CPL sample can be found from the demo repository, project named 'Call processing'. As usually, if you need to extend the created language further − add notational symbols, additional constraints, generators or by modifying dialogs and toolbars for modeling tools − you need to have MetaEdit+ Workbench or the evaluation version available from www.metacase.com.
- XML viewer or web browser for opening and validating the generated CPL scripts.

For further information about MetaEdit+, please refer to the 'MetaEdit+ User's Guide', 'MetaEdit+ Workbench User's Guide' or our web pages at http://www.metacase.com. For further information about CPL see Lennox et al, Call Processing Language: A Language for User Control of Internet Telephony Services at http://www.ietf.org/rfc/rfc3880.txt.

1 The Call Processing Language Example

The Call Processing Language (CPL) presents a graphical, domain specific modeling (DSM) language made for specifying Internet telephony services.

In this chapter we introduce the purpose of the CPL and the main concepts of the language. Chapter 2 explains how to access the existing models in MetaEdit+ and how to use the language by modeling a new telephone service that describes anonymous call rejection. Chapter 3 describes how the CPL example was implemented as a domain-specific modeling language and generator into MetaEdit+.

Please note that walking through the CPL example requires a basic knowledge how to use MetaEdit+. A good starting point to obtaining this knowledge is the Family Tree example in the 'Evaluation Tutorial'.

1.1 THE BASIC IDEA OF THE CALL PROCESSING LANGUAGE

The objective to have a DSM solution for CPL was being able to quickly and safely specify call processing services. Ideally, the service engineers could specify the telephony services by using directly the concepts they are already familiar with, without having to master or write XML manually.

To illustrate the CPL domain, let's look at some typical service products, which can be developed with the CPL language:

- Call forwarding if the receiver is busy or does not answer
- Rejecting all calls that originate from anonymous addresses
- Specifying a call to be forwarded to support personnel during office hours, while calls received outside office hours are forwarded to a voicemail service or web page.
- Proxying incoming calls to the receiver-registered station that best matches the media capabilities (e.g. video call) specified in the call request.

By implementing the CPL into MetaEdit+, we obtain an environment where a service engineer can create and modify the call service definitions and automatically generate quality, valid and well formed CPL scripts, ready to be executed in a call processing server.

1.2 AN EXAMPLE MODEL

Figure 1-1 illustrates a sample model of call processing service made with the DSM. The diagram specifies a call redirecting service where all incoming calls, which originate from "example.com", are redirected to the address "sip:jones@example.com". If there is no answer, a line is busy or a failure occurs, the call is redirected to a sub action. This 'subaction' provides an own model, which implements a redirecting service to the voicemail address. All

calls that originate from other addresses will be redirected directly to the same voicemail address.

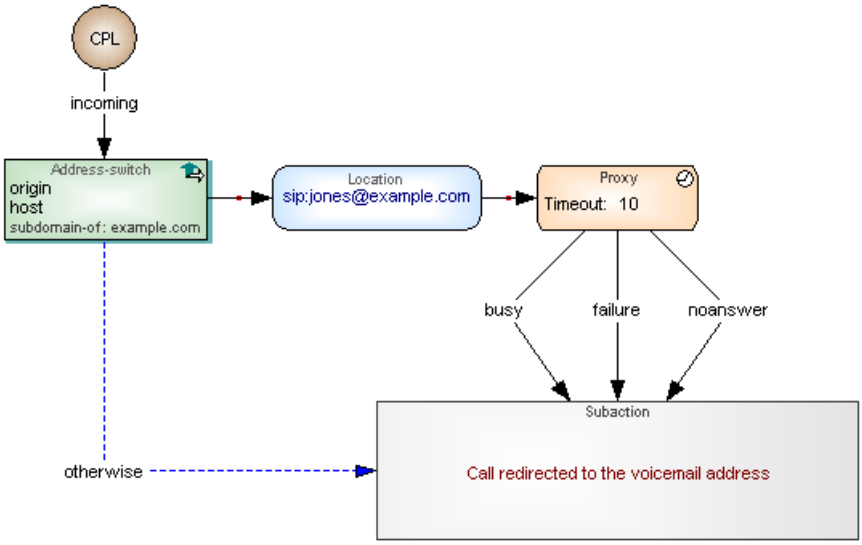
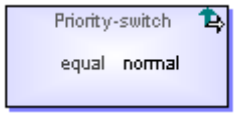


Figure 1-1. Incoming calls redirected

1.3 ABOUT THE CPL CONCEPTS

The CPL concepts include:

Language concepts	Representation of the concept
A root node represents the starting point for the call process. There can be only one start object in a model and there can be only one relationship starting from it.	
Switches represent the choices made when running the CPL implementation. There are five switch types in the language and each switch type has different kinds of properties to be entered. Switches can be 'address' (background green fountain fill color), 'language' (brown), 'priority' (purple), 'string' (orange) or 'time' (turquoise) related. In the symbol, the switch type is shown at the top and the choice arguments are represented as property values in the middle. In the sample shown on the right, the priority-switch has two properties: condition value, which is set to 'equal', and priorities value, which is set to 'normal'.	
The choice results are specified by connecting a switch to another modeling concept using a relationship; see 'default' and 'otherwise' paths.	
Location modifiers allow accessing the location data. Location modifier types are Location, Lookup or Remove-location. All of them have a light blue fountain fill	

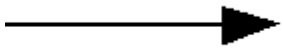
background color and their type is shown at the top. In the sample shown on the right, the location url 'sip:jones@example.com' is presented in the middle in blue.

Signaling operations are Proxy, Redirect and Reject. All of them have an orange background. Their type and related icon are presented at the upper part of the symbol. In the sample shown on the right, the rejection status, 'reject', is presented first, followed by the reason: 'I reject anonymous calls'.

Non-signaling operations are Log and Mail. These are presented with a yellow color. In the sample shown on the right, the mail object has an url 'sip:jones@email.example.com'.

Subaction is presented with a grey rectangle. The Subaction's name will be presented with red color. Each subaction has a detailed implementation which is described in a submodel. In MetaEdit+ that can be accessed by using the subaction's decomposition functionality, available by selecting the Subaction's popup menu.

Default path relationship specifies cases where a condition is met or next node is followed directly. Default path is drawn as a solid black line with an arrow head pointing to the next object to be called.



If default path has specific values, like after a Proxy or Lookup element, they will be presented in the middle of the line.

Otherwise relationship specifies an output for cases where a condition is not met. Otherwise is presented with a blue dotted line with 'otherwise' text in the middle.

To support the generation needs, the modeler can utilize a predefined CPL generator. This generator can be executed by clicking the editor's toolbar button named 'CPL'. After completion of the generation process, the generated file will be opened in your associated program, e.g. web browser.

2 Working with the CPL

In this chapter we discuss how to access the call processing language and how to work with it; first by playing around with existing models and then by creating a new call process service.

2.1 ACCESSING THE CPL EXAMPLE

To access the CPL examples, start MetaEdit+, login as usual into the demo repository and choose the 'Call processing' project from the project lists. When MetaEdit+ has completed the login procedure, the CPL examples can be accessed with the usual MetaEdit+ tools like the Graph Browser and the Diagram Editor.

All the CPL concepts are shown in the Diagram Editor's toolbar after opening a model. As you will see, the toolbar concepts are grouped; starting with the root node and continuing with switches, location modifiers, signaling operations and non-signaling operations.

2.2 PLAYING AROUND WITH THE CPL LANGUAGE

To start the tour of the CPL example, open any of the graphs listed in the Graph Browser. The first model in the list, 'Call redirected when they origin host example.com' illustrates a service for a call redirection based on the call's origin. Double-click it from the list to open it in the Diagram Editor. This model is also presented in Figure 2-1.

To access the properties of any model element, double-click the element in the diagram or in the integrated property sheet on the bottom-left of the Diagram Editor. You may also access operations related to each model item by first choosing the element and then opening its pop-up menu.

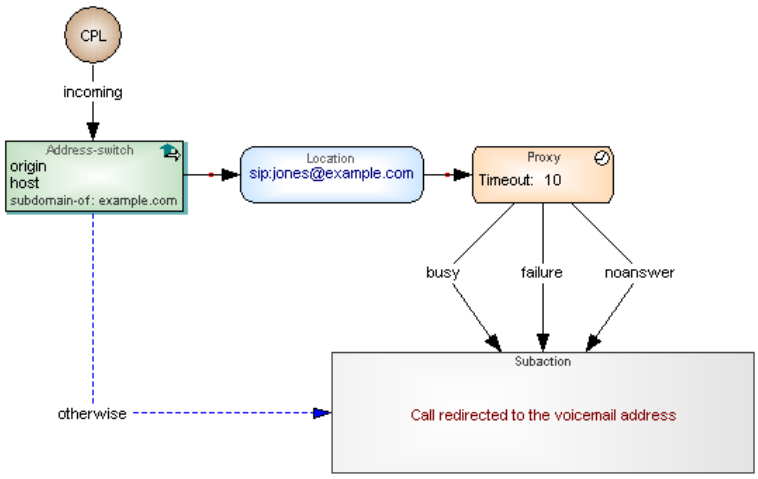


Figure 2-1. A sample service specification with the CPL language

As can be seen from the model, elements of the call process are linked with directed relationships. Normally, the relationships have no label that representing the default path. In cases involving the root, proxy and location objects, the relationship has a label that represents the given value triggering the relationship. The otherwise relationship is represented with a dashed blue relationship line, e.g. the relationship between the address-switch and the grey subaction object.

The details of the subaction are presented in a separate model. It can be accessed by double-clicking the subaction object while holding down the Ctrl-key. You may also first select the subaction in the diagram and then select **Decomposition…** from its pop-up menu. The pop-up selection also provides access to additional operations: for example for removing the link to the sub model or for replacing the current model with another one.

To invoke the code generator, simply press the CPL button in the Diagram Editor's toolbar. This will execute a generator for CPL and after the file is produced, the result is opened in the associated application like in a web browser as shown in Figure 2-2. This specification can be read by the CPL server that takes care of the actual call processing.

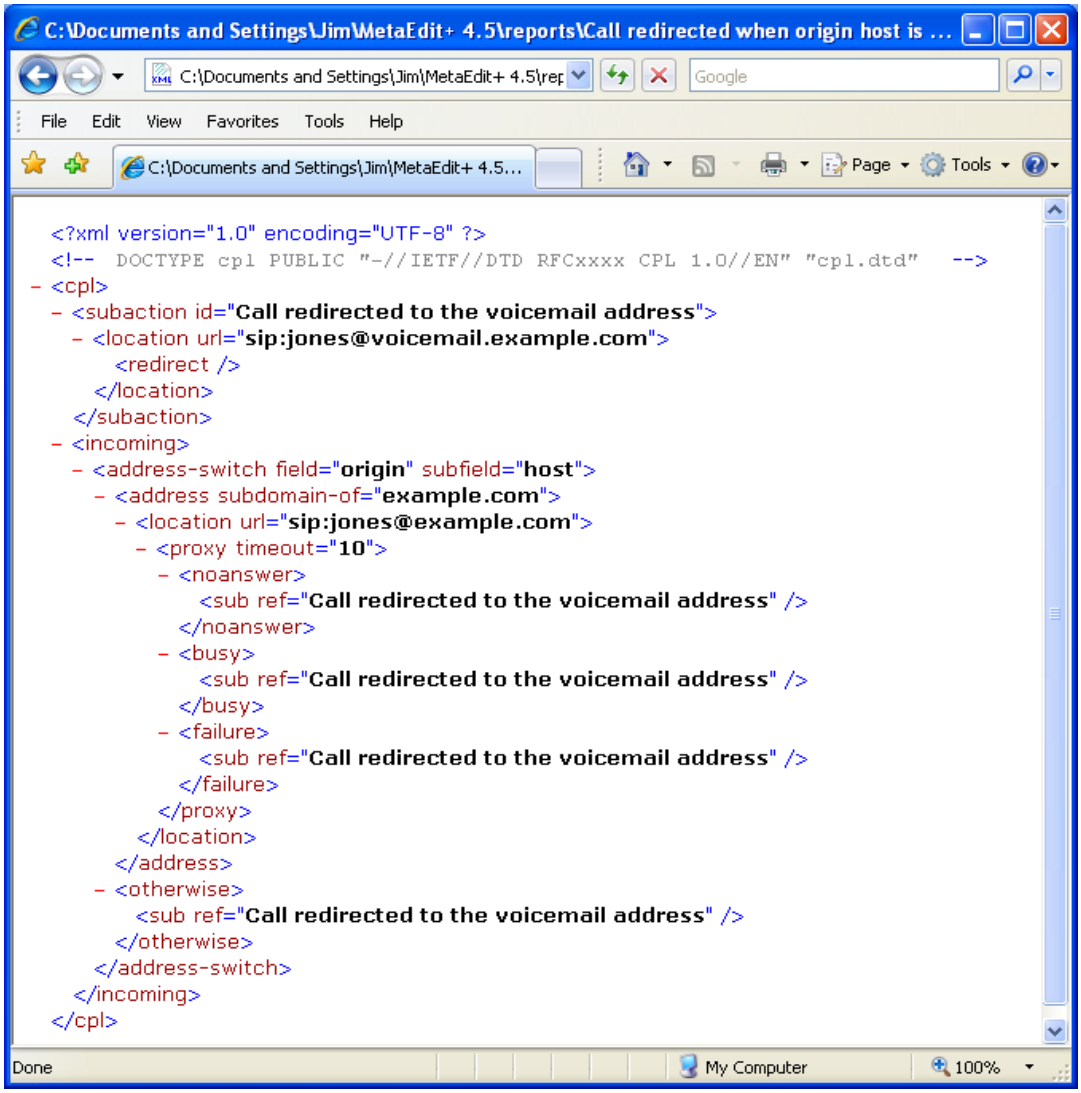


Figure 2-2. Generated call processing specification

2.3 CREATING A NEW MODEL

Next, we will use the CPL language to develop a new call processing service. We will start from scratch and make a new specification that rejects all incoming anonymous calls. Our call process specification uses CPL concepts directly, allowing us to create a new service description within a just few minutes.

2.3.1 Creating a new graph type

First, we need to create a new diagram for the anonymous call rejection. Click the **Create Graph** button in the main window or choose the same operation from the pop-up menu that can be opened from the middle window of Graph Browser. MetaEdit+ will ask you for the graph type you wish to use. Just select the 'Call Processing Language' from the list and press **OK.** Here, you may select the representation and editor for the new model. For this case of CPL and graphical language, the initial selection of Diagram is the one to choose.

Next, MetaEdit+ asks you to enter the name for the graph (Type 'Anonymous call rejection') and add other property values ('Author' and 'Description', which you may choose to leave empty for now) for the graph of call process specification. After pressing the **OK,** the dialog will close and an empty Diagram Editor will open.

2.3.2 Adding a new object to the model

Next, we specify the objects that our call process uses. We start with creating a root node from which the service script will start its execution. Choose the 'root node' button from the toolbar or select it from the Types menu and then click on the drawing canvas. Because the root node object does not have any properties, there will not appear a property dialog for entering details.

Next, we will specify the Address-switch object. In a similar manner as when selecting the root node previously, now select the 'address' object from the toolbar (next to root node) and then click on the drawing canvas. This opens a dialog that allows you to to specify the details of the created address-switch. For our case of call redirection, we need to specify a 'Field' value, 'Subfield' value and 'Address' value that will be compared during the call process execution. Possible values have been predefined and can be selected from the pull-down lists. The address value to be compared is entered as a string into the Address value field at the bottom.

After entering these properties, the dialog for specifying the 'Address' object should look like in Figure 2-3. Choose **OK** and close the dialog. This will add the created object into the diagram.

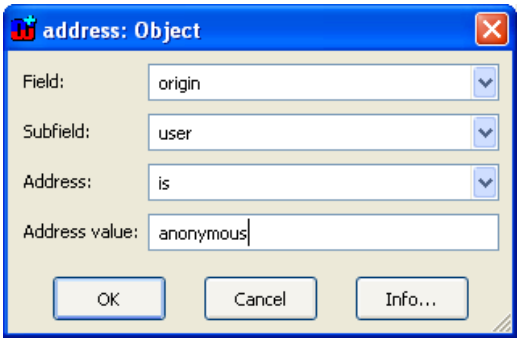


Figure 2-3. Dialog for adding the 'Address-switch' object

In a similar manner we can create a ‘reject’ object. Add the ‘reject’ object with property values ’reject’ for Status and for the rejection reason ‘I reject anonymous calls’. After adding all objects to the model, it should look like in the Figure 2-4.

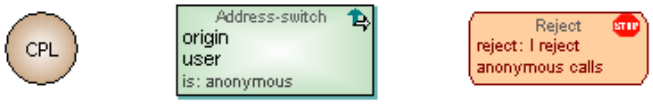


Figure 2-4. Modeling objects added to the service for call rejection

2.3.3 Creating bindings between objects

Next we will finish the service implementation by adding connections between the objects. You can do this by first selecting the root node object, then choose **Connect…** from its pop-up menu and finally click the address object as target for the relationship.

→ *For alternative relationship creation possibilities please see Diagram Editor chapter in MetaEdit+ User’s Guide.*

Creating a relationship will open a dialog to specify possible details. For the case of starting the call from the root element select ‘incoming’ as a session type from the pull-down list. Finally press **All OK** to close the dialog.

In a similar manner you can now also create the relationship from the Address-switch to the Reject object. During the relationship creation, double-click ‘default path’ relationship option provided by the dialog or select it from the list and then press **OK** button. The final model should now look similar to Figure 2-5.

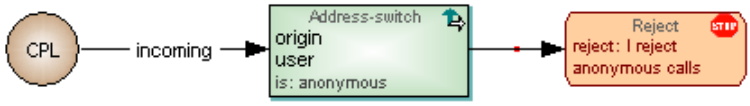


Figure 2-5. Anonymous call rejection service

To generate the CPL file from the graphical service specification, just press the CPL button provided by the toolbar. The CPL generator will go through the model, produce the CPL file and open it in the associated program.

3 How the CPL support was made

We have now described the language, its concepts, properties, connections and rules through the basic tasks of using the CPL language. Next, we will go through some key principles of the CPL architecture, its concepts and their implementation as a domain-specific modeling language into MetaEdit+.

3.1 CPL ARCHITECTURE

Architecturally, a call processing service is executed in a signaling server. Signaling servers are devices which relay or control signaling information. A signaling server also maintains a database of locations where a user can be reached. A call processing service makes its proxy, redirect, and rejection decisions based on the contents of that database. A CPL specification replaces this basic database lookup functionality; it takes the registration information, the specifics of a call request and other external information it wants to reference, and chooses the signaling actions to perform. Simply put, a CPL describes how devices respond to calls and how a system routs the calls.

The underlying objective for creating a DSM solution was to provide the ability to easily specify services, which are then generated and executed safely in a CPL server. Starting point for the modeling language development was the idea of using graphical models. The specification of the language for defining services was available as an XML schema (Lennox et al. 2004).

3.1.1 Location object

The definitions of the domain concepts, needed for modeling language construction could be taken directly from the XML Document Type Definition (DTD). Let's take an example: the 'location' concept in CPL. The specification of the 'location' can be found from the CPL DTD as follows:

```
<!ENTITY % Clear 'clear (yes|no) "no"'>
<!ELEMENT location (%Node;)>
<!ATTLIST location
      url CDATA #REQUIRED
      priority CDATA #IMPLIED
      %Clear;
>
```

This piece of DTD specifies that the 'location' concept has three properties:

- The url of the address, a string value, which will be added to the script's location set. The Url value is mandatory (#REQUIRED) , which is checked via a regular expression specification in the property tool of MetaEdit+.

- priority, which specifies a priority for the location. Its value is a floating-point number between 0.0 and 1.0 or is just empty. In the metamodel this checking is done again by using a regular expression. Figure 3.1 illustrates this using the Property Tool for defining Priority.

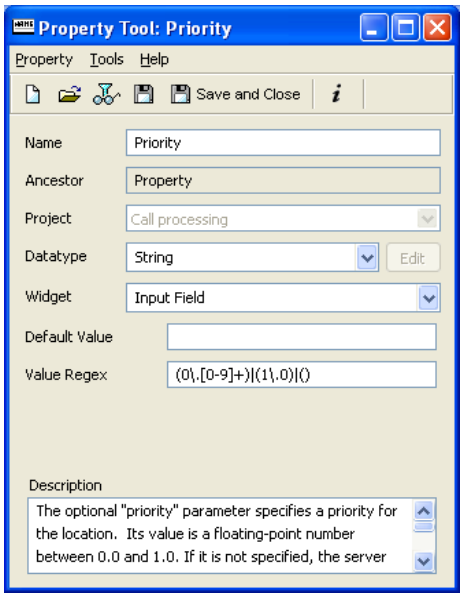


Figure 3-1. Priority's property definition

- clear value, which specifies whether the location set will be cleared before adding a new location to it. In MetaEdit+ the property definition has predefined pull down list with values 'yes' or 'no', with 'no' set as the default.

What the above looks in the modeling language? The DSM follows a pattern, where all similar kind of language concepts are represented with the same shape, size and coloring schema. For example location modifiers, namely 'location', 'lookup' and 'remove-location', have the same rounded rectangle format, blue background color and an operation-specific symbol in the top-right corner. Figure 3-2 illustrates the location symbol definition.

Figure 3-2. Location symbol

The 'Location' symbol consists of four main elements: Grey text for the object types at the top (here 'Location') and dark blue 'Url' property value in the middle. The red line around the object and the crosshair in the middle specify the connectable area. The main symbol has a fountain fill background color, starting with a white color in the middle fading to light blue towards the border.

3.1.2 Other metamodel specialties

The language provides no concept for specifying the end of the call process. The execution of the service ends when the last element in the flow is reached and performed. Then a CPL server performs the action and the service ends. Both 'Redirect' and 'Reject' objects immediately terminate the call processing execution so these concepts were defined to have just incoming flows – i.e. they form leaves of the tree.

The ‘Proxy’ and ‘Lookup’ objects are special cases, where the default path relationship must have a named value. In the proxy case, there is a possibility to choose the default path condition from predefined values of ‘redirection’, ‘busy’, ‘default’, ‘failure’ and ‘noanswer’. In the lookup case, the choices are ‘success’, ‘notfound’ and ‘failure’. If any of these values is used more than once for a given object, a checking report will provide an error and a link to the object in the model that causes the error.

As many services have common functionality they could be treated as re-usable service components to be called by other actions. For this purpose, a ‘subaction’ concept was defined into the language: it enables an action to decompose into another model, which is treated as a sub-model. As the subaction could be defined by using the same language concepts as the main CPL definition there is no need to have a separate language for defining the subactions. In the metamodel this was achieved by defining a decomposition link from a subaction object to the Call Processing Language graph.

3.2 CPL CONSTRAINTS

Along with the identification of the modeling concepts, many of the constraints and model correctness rules were identified. Where possible they were defined to be part of the metamodel so that they are checked instantly during design time. In MetaEdit+, such constraints can be defined by using the graphical metamodeling language (see Graphical Metamodeling Example for details) or alternatively with the in-built metamodeling tools of MetaEdit+ Workbench.

Figure 3-3 illustrates some constraints defined for the CPL language. Constraints like ‘there can be only one root node in the diagram’ or ‘there may be only one default path from switches’ are defined by choosing constraint types and related language concepts in the Constraints Tool of the MetaEdit+ Workbench.

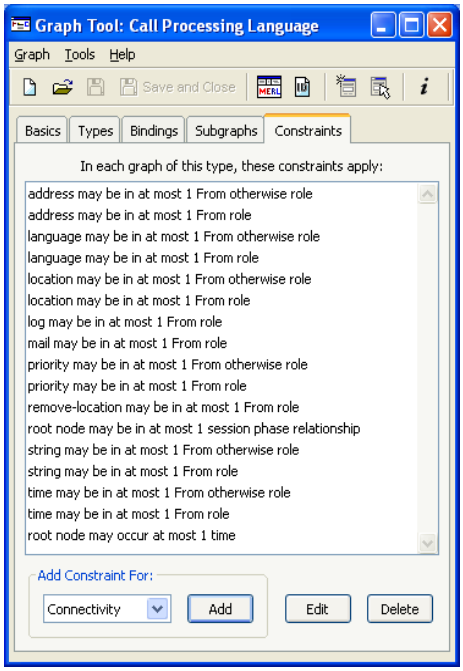


Figure 3-3. CPL metamodel specific constraints

In addition to rules and constraints that are checked at modeling time, model checking can also be performed by using the generator. In this case, the generator analyzes the rules that can’t be checked or are not sensible to check after each modeling action. The checking

generator can be launched either by pressing the 'Check' button from the editor's toolbar or alternatively by embedding its output in a 'comment' object in the model itself.

The 'comment' object lists the results of the same checking generator, like cyclic structure found in the diagram, if several overlapping values from proxy or lookup objects were found, if there is a non-connected object etc. Another useful way to find the cyclic structures is to select **Graph | Layout** in the Diagram Editor, followed by selecting all the relationships from the list and pressing **OK**. After that, MetaEdit+ will open a dialog where the cyclic paths are listed. When a cyclic path is selected, it will be highlighted in the diagram. If no cyclic paths are found, the layout algorithm creates a new layout. The original layout can be restored by performing an Undo action.

3.3 CPL GENERATOR

Defining generators to produce XML is quite a straightforward process: elements in a model and their connections are described by XML tags. In the beginning of the XLM document the settings are defined. Then the generator starts by going through all the subactions of the service specification. The generator visits each object in the model and calls a generator module for that element's type. After that, the generator crawls to the next object via the 'default path' relationship. In a similar manner, the generator goes on until it founds the object where no outgoing relationships were made. 'Otherwise' relationships are always generated after the generation of the default path has been completed.

The CPL generation did not require any framework code to make model-based code generation possible. As the service specification is made based on the CPL specification the schema defines pretty much what the generated code should look like. You can also modify the generator or create new ones by opening the Generator Editor by choosing **Edit Generators** from the **Graph** menu. For more details on defining generators, please see MetaEdit+ Workbench User's Guide.

4 Conclusion

In this example we have demonstrated working with the call processing language. With the CPL language you can design the basic services for call processing servers. The CPL schema concepts are mapped almost one-to-one to the DSM concepts. CPL is implemented as any other modeling language in MetaEdit+. It is completely open and thus it can be freely extended to cover additional requirements of call handling like VoiceXML or SIP services. You are welcome to extend the CPL language as well as the generators further.