Version 5.5

The Call Processing Language Example

MetaCase Document No. CE-5.5

First Printing, 2nd Edition, August 2018

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä
Finland

Tel: +358 400 648 606
E-mail: info@metacase.com
WWW: http://www.metacase.com

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 directly using 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.

To explore 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, in the 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) example presents a graphical, domain-specific modeling (DSM) language made for specifying Internet telephony services. It was created based on the existing CPL XML schema, and thus also serves as an example of how DSM can provide an intuitive graphical front end to an existing textual format.

In this chapter we introduce the purpose of 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 for 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 basic knowledge on how to use MetaEdit+. A good starting point for 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 to be able to quickly and safely specify call processing services. Ideally, service engineers could specify telephony services by directly using 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 CPL in MetaEdit+, we obtain an environment where a service engineer can create and modify the call service definitions and automatically generate high 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 a call processing service made with the DSM language for CPL. 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 'subaction'. This subaction provides its own service model, which implements redirection to the voicemail address. All calls that originate from other addresses will be redirected directly to the same voicemail address.

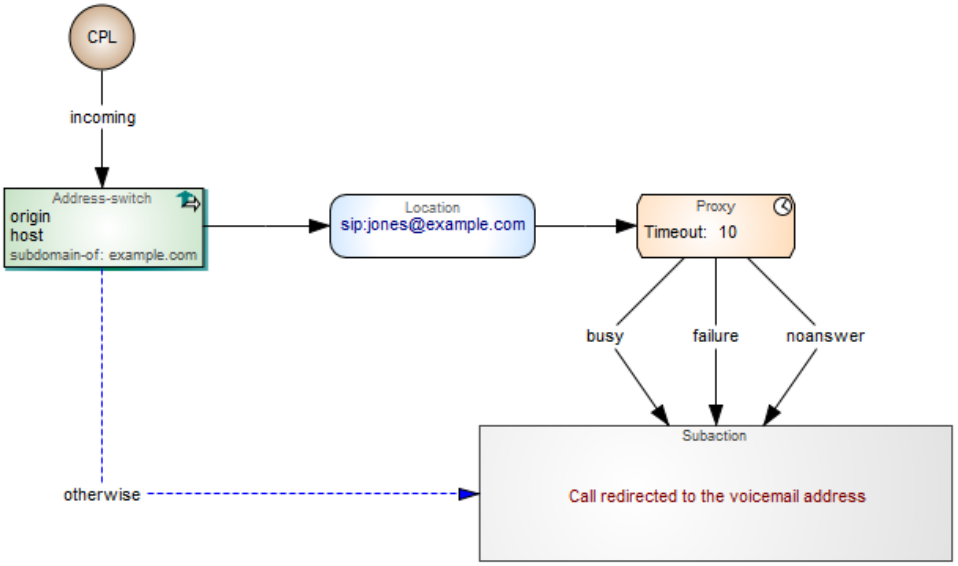
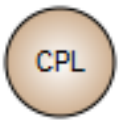


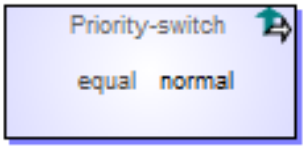
Figure 1-1. Incoming calls redirected

1.3 ABOUT THE CPL CONCEPTS

The CPL concepts include:

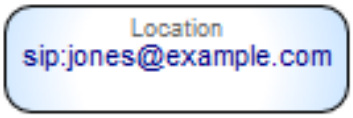
Description	*Representation*
A root node represents the starting point for the call process. There can be only one start object in a model and there can be only one relationship starting from it.	

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 based on ‘address’ (background color: green fountain fill), ‘language’ (brown), ‘priority’ (purple), ‘string’ (orange) or ‘time’ (turquoise). 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 example 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’.

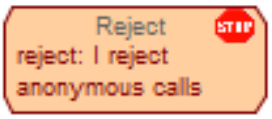


The choice results are specified by connecting a switch to another modeling concept using a relationship; see ‘default’ and ‘otherwise’ paths.

Location modifiers offer access to 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 example shown on the right, the location URL ‘sip:jones@example.com’ is shown in the middle in blue.

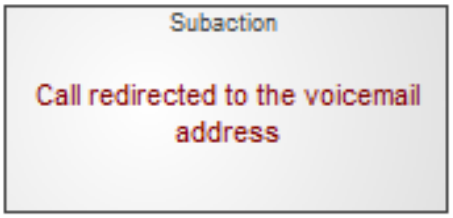


Signaling operations are Proxy, Redirect and Reject. All of them have an orange background. Their type and related icon are shown at the top of the symbol. In the example shown on the right, the rejection status, ‘reject’, is presented first, followed by the reason: ‘I reject anonymous calls’.



Non-signaling operations are Log and Mail. Both have a yellow background. In the example shown on the right, the mail object has a URL ‘sip:jones@email.example.com’.

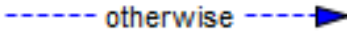
Subaction is shown as a grey rectangle. The subaction’s name will be shown in red. Each subaction has a detailed implementation which is described in a subgraph. In MetaEdit+ the subgraph can be opened by Ctrl-double-clicking the subaction object, or from its pop-up menu.



The default path relationship specifies cases where a condition is met or the next node follows directly. The default path is shown as a solid black line with an arrowhead pointing to the next object to be called.

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

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



The CPL example includes a generator to produce CPL scripts in XML format. 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. a web browser.

2 Working with CPL

In this chapter we discuss how to access the call processing language and how to work with it. We first play around with existing models and then create a new call process service.

2.1 ACCESSING THE CPL EXAMPLE

To access the CPL examples, start MetaEdit+ and login as usual into the demo repository, choosing the 'Call processing' project from the project lists. The CPL examples can then 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 origin host is 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 shown in Figure 2-1.

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

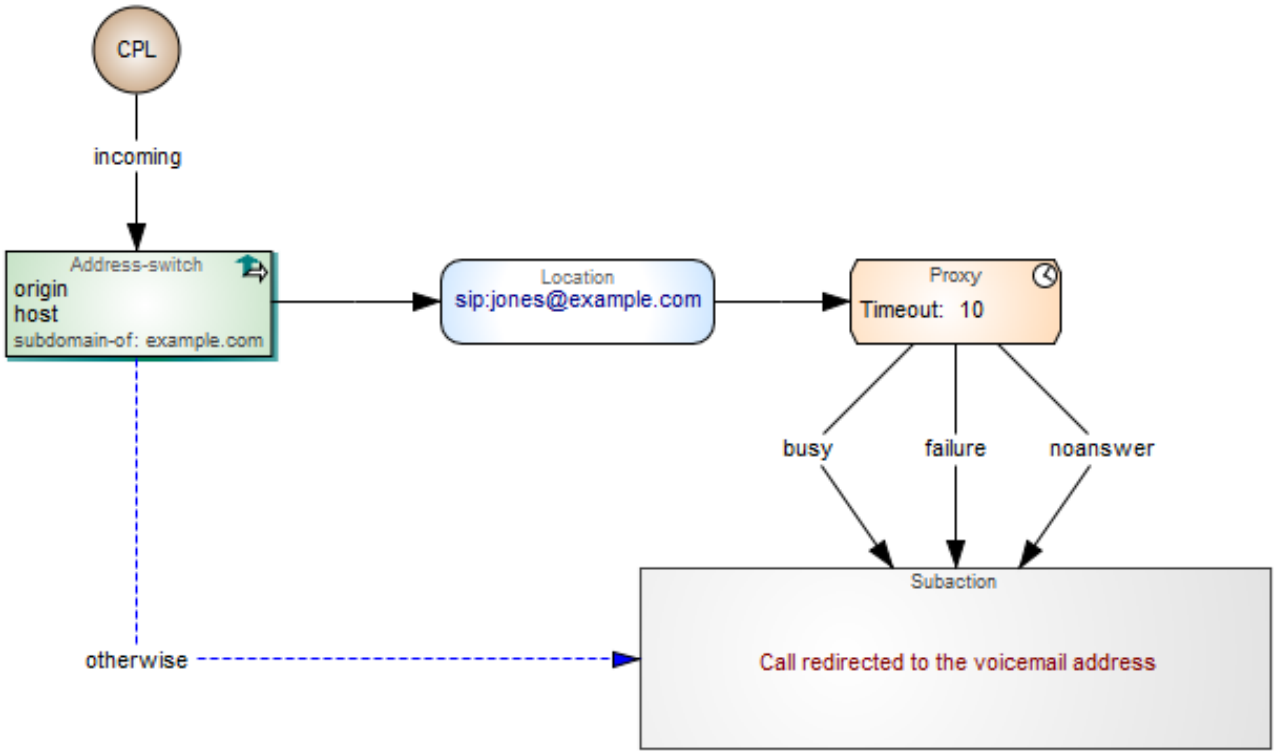


Figure 2-1. A sample service specification with the CPL language

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

The details of the subaction are presented in a separate graph. 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 **Open Subgraph** from its pop-up menu. The pop-up menu also provides access to additional operations: for example for managing the link to the subgraphs.

To invoke the code generator, simply press the CPL button in the Diagram Editor's toolbar. This will execute a generator for CPL and open the resulting file in the associated application, e.g. 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.

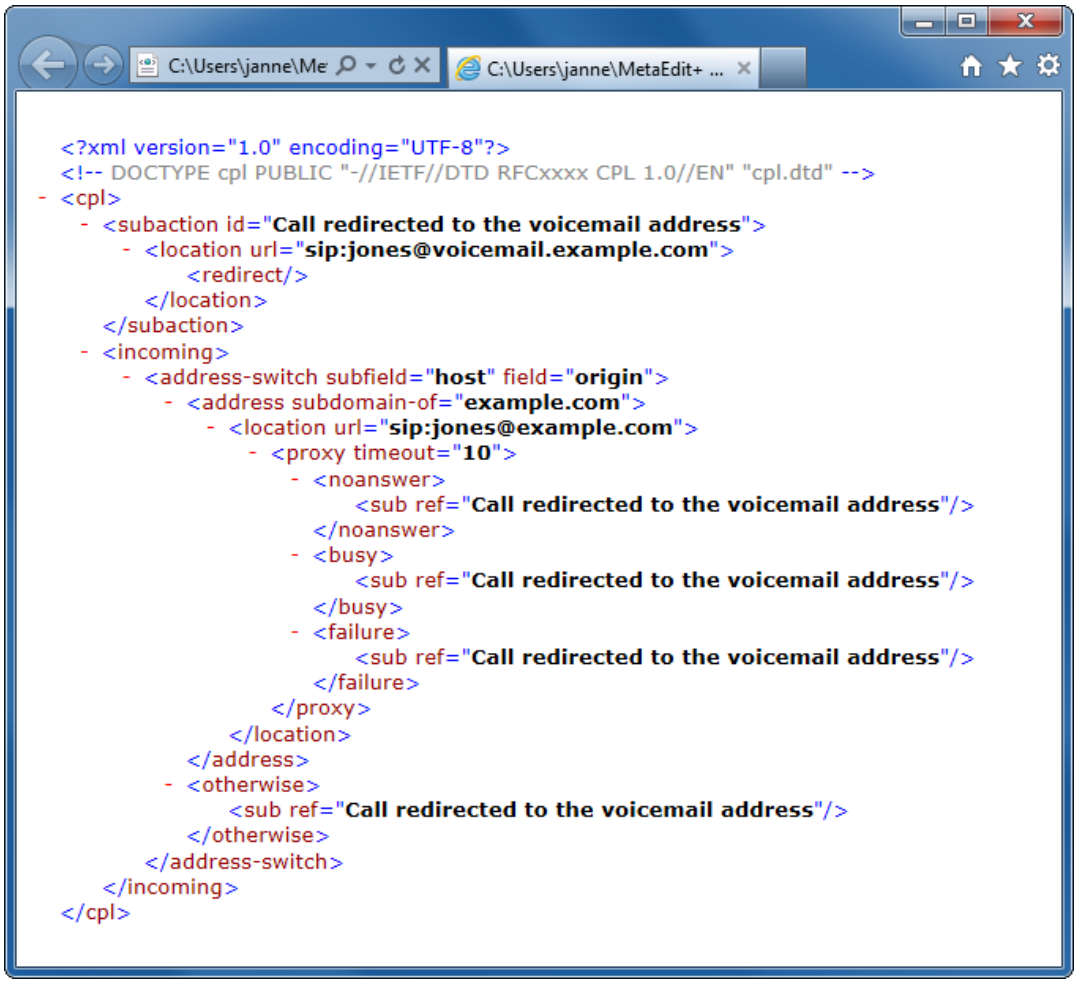


Figure 2-2. Generated call processing specification

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 anonymous incoming calls. Our call process specification uses CPL concepts directly, allowing us to create a new service description within just a few minutes.

2.3.1 Creating a new graph

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 pane of the 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.** In the same dialog you may select the representation and editor for the new model: the default initial selection 'Diagram' is what we want here.

Next, MetaEdit+ asks you to enter the name for the graph, e.g. 'Anonymous call rejection'. You can leave the 'Documentation' property empty for now. After pressing **OK,** the dialog will close and a Diagram Editor will open on the new empty graph.

2.3.2 Adding a new object to the model

Next, we specify the objects that our call process uses. We start by creating a root node from which the service script will start its execution. Click the 'root node' button on the type toolbar, or select it from the Types menu, and then click on the drawing canvas. As the root node object does not have any properties, no property dialog will appear.

Next, we will specify the Address-switch object. In a similar manner as when creating the root node previously, click the 'address' object type on the toolbar (next to root node) and then click on the drawing canvas. This opens a property dialog that allows you to specify the details of the 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. Legal values for these first three properties can be selected from their pull-down lists. The address value to be compared is entered as a string, 'anonymous'.

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

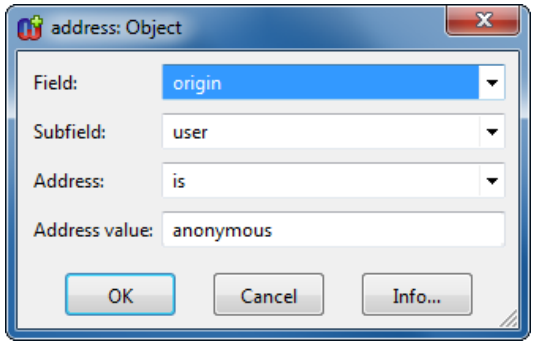


Figure 2-3. Dialog for adding the 'Address-switch' object

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 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 connecting the objects with relationships. To create the first relationship, select the root node object, choose **Connect…** from its pop-up menu, and click the address object as the target for the relationship.

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

Creating a relationship will open a dialog to specify possible details for the relationship: .in this case, select 'incoming' as the 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. As there is more than one possible relationship type between these objects, you

will be prompted to pick the desired one from a dialog: double-click ‘default path’. The final model should now look similar to Figure 2-5.

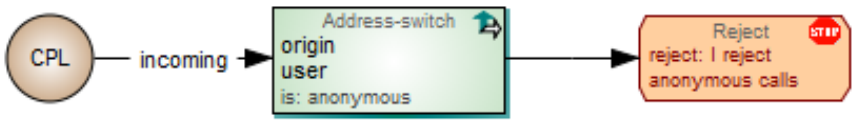


Figure 2-5. Anonymous call rejection service

To generate the CPL file from the graphical service specification, just press the CPL button on 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 in 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 specification 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. The 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 in the modeling language could be taken directly from the XML Document Type Definition (DTD). Let's take an example: the 'location' concept in CPL. The specification of '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) in the CPL DTD; this can be checked with a regular expression on that property type in the MetaEdit+ metamodel.
- 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 again performed

using a regular expression. Figure 3.1 illustrates this using the Property Tool for defining Priority.

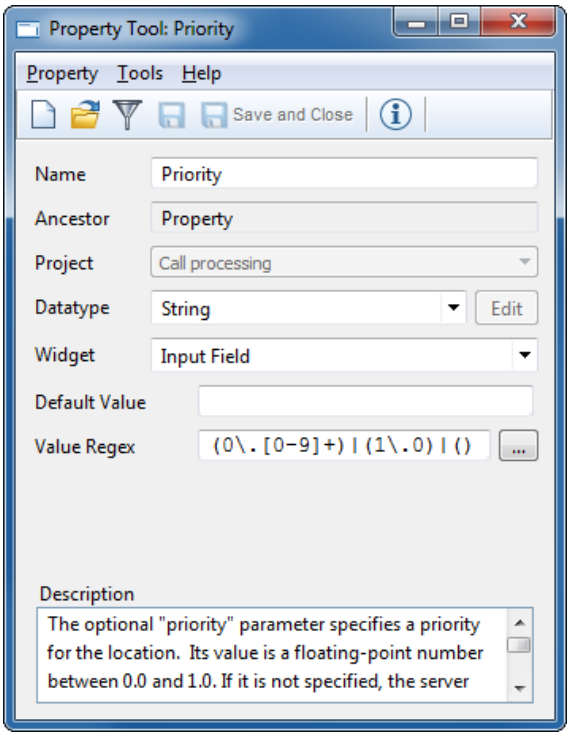


Figure 3-1. Priority's property definition

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

What should a 'location' look like in the modeling language? The CPL DSM languages 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 icon in the top-right corner; remove-location has an 'X', lookup has a magnifier and location has none. Figure 3-2 illustrates the location symbol definition.

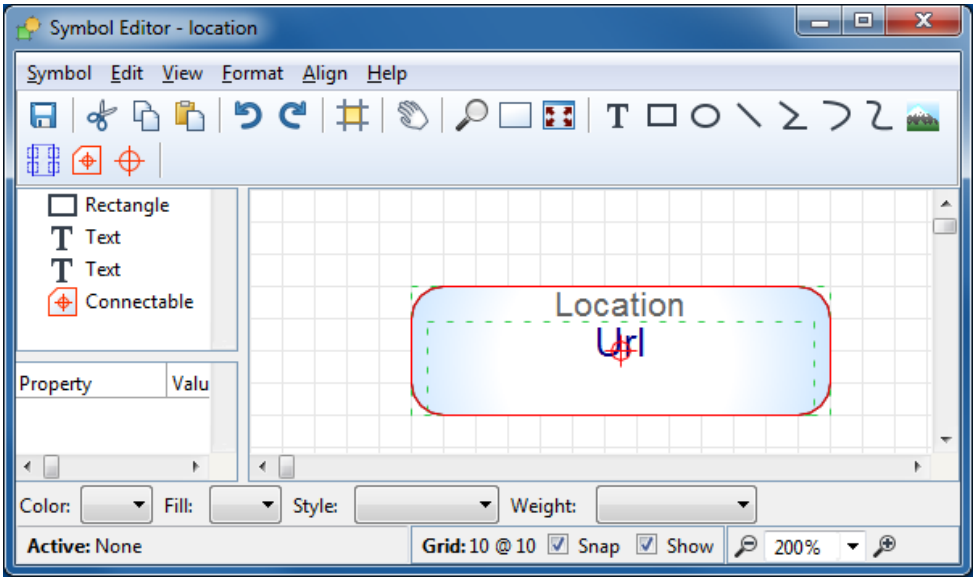


Figure 3-2. Location symbol

The ‘Location’ symbol consists of four main elements: The main symbol has a fountain fill background color, starting with a white color in the middle fading to light blue towards the border. The type name, ‘Location’, is shown in gray at the top, and the ‘Url’ property value is shown in blue in the middle. The red line around the object and the crosshair in the middle specify the connectable area, which is only shown when connecting objects via relationships: role lines will point towards the crosshair and stop at the red outline.

3.1.2 Other metamodel details

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.

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 show the error with a link to the object in the model that causes the error.

As many services have common functionality, this can be treated as reusable service components to be called by other actions. For this purpose, a ‘subaction’ concept was defined in the language: it specifies a link to a subgraph, which is treated as a call to the operations in the subgraph. The subgraph uses the same language concepts as the main CPL definition; there is no need to have a separate language for defining the subactions’ contents. In the metamodel this link is defined in the Call Processing Language graph type, as a decomposition subgraph link from the ‘subaction’ object type to the Call Processing Language graph type.

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+ Workbench, such constraints can be defined in the Graph Tool, or by using the graphical metamodeling language (see Graphical Metamodeling Example for details).

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.

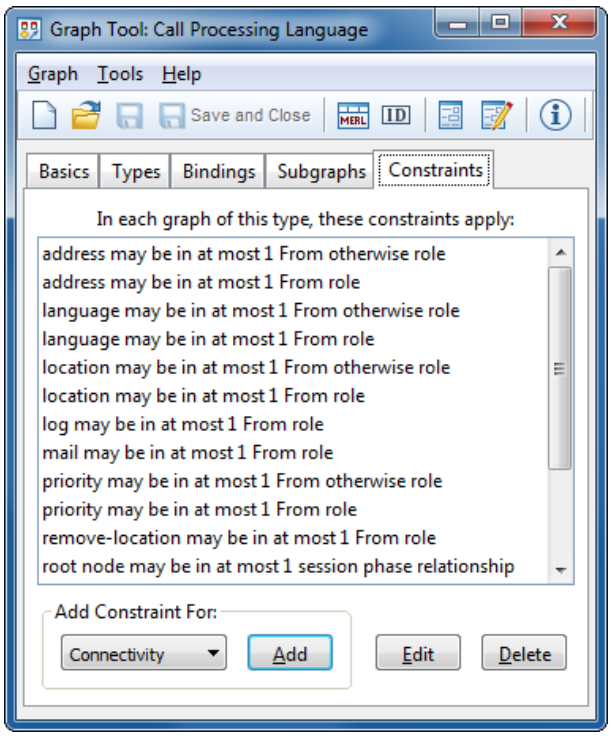


Figure 3-3. CPL metamodel specific constraints

In addition to rules and constraints that are checked at modeling time, model checking can also be performed by using the generator. Generators are used for rules that cannot sensibly be checked 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. At the bottom of the diagram, the LiveCheck pane shows the output of the __LiveCheck generator, executed after every modeling operation in the editor.

The 'comment' object lists the results of the same checking generator: cyclic structures found in the diagram, overlapping values from proxy or lookup objects, unconnected objects etc. Another useful way to find 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 from the list, 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 elements. At the top of the XML document the XML header settings are defined. Then the generator starts by going through all the subactions of the service specification, producing output for their subgraph objects in a similar way to the main body of the generator. The main body of the generator starts at the root node and visits each object in the model, calling a subgenerator for it named after that object's type. After that, the generator crawls to the next object via the 'default path' relationship, continuing recursively there, and then similarly for any 'otherwise' relationship.

The CPL generation is thus rather simple, as is often the case when making a DSM language as a front end for an XML schema. 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 the 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 like any other modeling language in MetaEdit+. It is completely open and thus it can be freely extended to cover additional requirements for call handling, such as VoiceXML or SIP services. You are welcome to extend the CPL language as well as the generators further.