### CHAPTER 5

## IP TELEPHONY AND CALL PROCESSING

In this chapter we describe DSM solution made for specifying Internet telephony services. Recently multiple protocols, such as SIP (Rosenberg et al., 2002) and H.232 (ITU, 2003), have been defined to provide a session initiation protocol for telephony over IP networks. These protocols also offer the possibility to decentralize the control of user-specific call processing services. In traditional telephony systems, networkbased services were created only by the service providers, such as operators. Service creation has required special knowledge of the telephony system in use and use of a variety of often proprietary tools, and most importantly many of the user-specific service customizations were simply not available. Internet telephony changes this as protocols are open and allow even telephony users to define their own customized services. Here, a DSM solution is created to allow easy specification of call processing services using telephony service concepts.

5.1 INTRODUCTION AND OBJECTIVES

One of the major changes IP-based technology offers for telephony is call processing applications. Many of the telephony services typically depend on a user’s status and their creation does not necessarily require intermediary organizations. In fact, the cost of their production would be less if they were made closer to their user. To illustrate the

Domain-Specific Modeling: Enabling Full Code Generation, Steven Kelly and Juha-Pekka Tolvanen Copyright # 2008 John Wiley & Sons, Inc.

97

IP TELEPHONY AND CALL PROCESSING

application development, let us look at some of the typical telephony services illustrated by Lennox and Schulzrinne (2000):

. Call forwarding if the receiver is busy or does not answer: For instance, when a new call comes in, the call should ring at the user’s desk telephone. If it is busy, the call should always be redirected to the user’s voice mail box. If, however, there is no answer after four rings, it should be redirected to the user’s voice mail unless it is from a supervisor, in which case it should be proxied to the user’s cell phone if it is currently registered.

. Information address service: An example service could be made for a company that advertises a general “information” address for prospective customers. When a call comes in to this address during office hours, the caller should be given a list of the people currently willing to accept general information calls. If it is outside of working hours, the caller should get a web page indicating what time they can call.

. Intelligent user location: For example, when a call comes in, the list of locations where the user has registered should be consulted. Depending on the type of call (work, personal, etc.), the call should ring at an appropriate subset of the registered locations, depending on information in the registrations. If the user picks up from more than one station, the pickups should be reported back separately to the calling party.

. Intelligent user location with media knowledge: One service could be that when a call comes in, the call should be proxied to the station the user has registered from whose media capabilities, such as video call, best match those specified in the call request. If the user does not pick up from that station within a specified number of attempts, the call should be proxied to the other stations from which the user has registered, sequentially, in order of decreasing closeness of match.

The call processing framework and language (Lennox et al., 2004) present an architecture to specify and control Internet telephony services such as those described above. The language part is of interest to us here. The purpose of the Call Processing Language (CPL) is to be powerful enough to describe a large number of services and features but at the same time to be limited in its power so that it can run safely on Internet telephony servers. The limited scope makes sure that the CPL server’s security will be ensured. Looking from the DSM creation side, the domain is well restricted and the requirement of complexity hiding is highly relevant. Also, generators offer obvious advantages here as quality and correctness of services are of great importance. Nobody wants to miss a call just because the service definition had errors.

A computational model of a CPL specification is a list of condition and action pairs: if the system condition matches a condition specified in a service, then a corresponding action or actions are performed. A typical system condition in CPL is, for example, that a call arrives and the line is busy or it is a certain day of the week. Example actions redirect the call to the user’s mobile phone or reject the call.

### INTRODUCTION AND OBJECTIVES

If conditions are not specified in the CPL specification, then the server’s standard action is taken.

5.1.1 CPL Architecture

Architecturally, a call processing service is executed in a signaling server. Signaling servers are devices that relay or control signaling information. In the SIP world (Rosenberg et al., 2002), examples of signaling servers are proxy servers and redirect servers. A signaling server also normally 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. To put it simply, CPL describes how devices respond to calls and how a system routes calls.

Services in CPL are normally associated with a particular Internet telephony address. When a call arrives at a signaling server that is a CPL server, the server associates the source and destination addresses specified in the request with its database of CPL services. If one matches, the corresponding CPL service is executed.

At this point, we should emphasize that creation of call processing services is not bound to any specific user type or organization. Services can be created by different kinds of people as follows:

. An end user can make or modify a service by defining a CPL script and uploading it to a server. This type of user is typically a nonprogrammer.

. A third-party service provider can create or customize services for its clients.

. The administrator of an IP telephony server can create services or describe policy for the servers they control.

. Middleware software could create and modify the services, and CPL would be the back-end for their execution. Here, a CPL service could then be a component for other services.

5.1.2 Why Create a DSM Solution for CPL?

The underlying objective for creating a DSM solution was to allow services to quickly and reliably specified and generated to be executed in a CPL server. The DSM language was not targeting end users as their service needs were considered to be solved better by choosing predefined services. The target users of the language were thus service providers and administrators. The actual service specification for the server is specified in XML. The definition of the language for defining services was available as an XML schema (Lennox et al., 2004)—a common starting point for defining modeling languages. It is not a bad starting point as many domain concepts are already identified. However, and as Lennoxet al. also acknowledge, XML schemas alone are not adequate to specify correctness of specifications made with the language.

5.2 DEVELOPMENT PROCESS

Development of the language started from the interest of one company to find an easy way to specify CPL services. Writing the services in handwritten XML was considered difficult, error-prone, and as easily leading to internally inconsistent specifications. Having generators produce the specifications in XML was anticipated to give significant productivity and quality improvements.

The starting point for modeling language development was the idea of using graphical models, already suggested in the CPL specification. The language notation was briefly outlined there with the example model shown in Fig. 5.1. Although only some modeling concepts were presented in this sketch example, the idea of using flow models as a computation model was a clear starting point. The rest of the CPL specification, however, focused on domain concepts and their semantics, not on modeling language concepts.

DSM was soon discovered to be of interest for two different kinds of companies: an operator and a manufacturer of telecom equipment. The actual language specification was largely the same because the starting point, the domain of call processing, was the same. The differences were in the process of use and in the extension and integration with other domains. The operator wanted to integrate CPL specifications with other designs, such as specifying voice messages during the call using VoiceXML. The equipment manufacturer wanted to generate Java code based on the SIP framework used in its product platform. This code would configure hardware along with CPL. While when discussing the DSM solution creation in detail, we focus on the common and publicly available CPL specification. Possibilities for extension and integration with other domains are, however, also described along with the case.

The first working draft of the DSM language and generator was made in 2 days. The draft covered about 80% of the language concepts. Implementing the generator took another 2 days and most of time that was spent on handling special cases of generating default path code. More about this is discussed later. The first draft was made by a

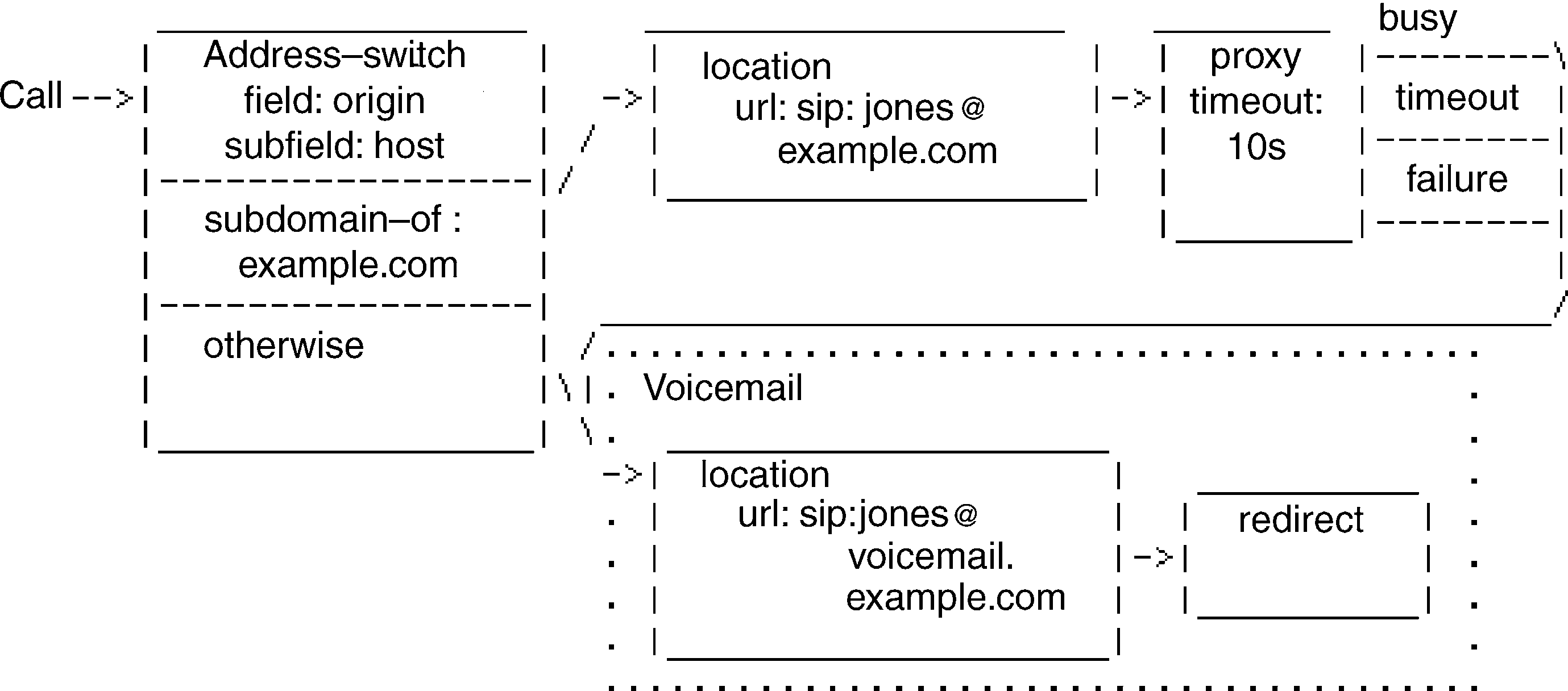


FIGURE 5.1 Idea for the modeling language notation (Lennox et al., 2004)

consultant using the public specification and the rest of the language was completed by the operator. The quick development time is largely explained by having a formal and adequately detailed specification of the domain: the XML schema. The specification also provided examples of CPL service specifications that served as test cases for the DSM solution. About 20 different kinds of services were made and compared against these examples. This testing was largely done by the external consultant using normal XML validation tools, as the CPL servers were not yet available during language creation. On the basis of the feedback, some minor changes to the language were made and model checking was implemented to support the specification process.

5.3 LANGUAGE FOR MODELING CALL PROCESSING SERVICES

DSM for CPL was implemented in small increments using existing IP telephony service specifications as test cases. These test cases were specified by the customer and were partly available in the CPL specification itself (Lennox et al., 2004). In fact, the implementation of the language was done in much the same order that the CPL language was described in this original CPL specification documentation. After adding the first few concepts to the modeling language, the generator was extended to cover the same concepts. This allowed immediately making an equivalent service specification with the DSM language and producing the CPL script for comparison with the relevant test cases. This process was followed until all the modeling concepts, and thus the full CPL specification, were handled.

The structure of the languagewas also taken directly from the CPL framework. The XML schema provided the language concepts and many of the constraints. A call processing action is structured as a tree that describes the operations and decisions a telephony signaling server performs when a call is made. Each node in the tree has parameters that specify the precise characteristics of the node. The nodes usually also have outputs, which depend on the result of the decision or action. The starting point for the language could thus be based on already established domain-specific concepts. The names used by the XML schema were followed exactly as this made the task of the generators easier. The naming needed for the generated output could be used straightforwardly in the modeling language. Thus the generator could be simpler and there would be no need to translate the names of the modeling concepts to those needed in XML. Later, during generator implementation, this was discovered to not be fully true as legal naming in XML required some string manipulation (see Section 11); for example, spaces are not allowed in the as names and special characters given in the model can be treated as XML tags.

Often, the specification, service examples, and XML schema used different names for the same domain concepts, such as “signaling actions” and “signaling operations” that both mean the same thing. This is a typical situation when the language is not formally defined and there is no reference implementation. Thus, creators of a DSM solution need to make choices and have a major influence on naming policies and establishing a vocabulary.

5.3.1 Modeling Concepts

The modeling language was constructed based on a flow model, with a special root element to start the script. After the root element, other modeling concepts were call processing concepts that usually pointed to the next stage in the call path. This service flow forms a tree structure and ends when the last element in the chosen path is reached. Constraints were given to prevent cyclic structures as discussed later in Section 5.3.2. The modeling concepts were divided into four categories based on the classification of their nature already found in the CPL framework: signaling operations, switches, locations, and nonsignaling actions.

Signaling Actions Signaling actions cause events that a CPL server can perform. A server can proxy a call setup, respond with redirecting information, or reject a call setup. Depending on the signaling action, different properties must be defined. To guide this, CPL specifies its own attributes for each action type. These are added to the modeling language as their own constructs. The concept definition needed for the modeling language construct could be taken directly from the XML Document Type Definition (DTD). Consider the following case of Proxy definition in the DTD:

<!-–Thedefaultvalueoftimeoutis’’20’’ifthe<noanswer> outputexists.--> <!ATTLISTproxy

|  |  |
| --- | --- |
| timeout | CDATA #IMPLIED |
| recurse | (yes|no) ’’yes’’ |
| ordering | (parallel|sequential|first-only)’’parallel’’ |

>

This piece of DTD specifies that a proxy concept has three properties. The same concepts could be added to the modeling language concept too:

. The time-out interval specifies how long to wait before giving up the call attempt, that is, a number of seconds. There is no default value to be used in the language for this property, but in case a call is not answered, a default value of 20 second is used. This default value is already set in the CPL server.

. Recurse specifies if the server should automatically make further call attempts to telephony addresses when responses are returned from the server that initiated the call. In the language, this was defined as a list of two values (yes, no) with a default value of the parameter (yes) chosen.

. Ordering specifies in which order the user locations should be tried. In CPL, a location set is maintained and there are three ordering possibilities. These were implemented to the modeling language similarly to recurse: parallel, sequential, and first-only are available as a list from which one must be chosen. The default value, parallel, is already chosen as a prefilled value when a proxy object is added in a model. These choices minimize the modeling work and reduce typing errors.

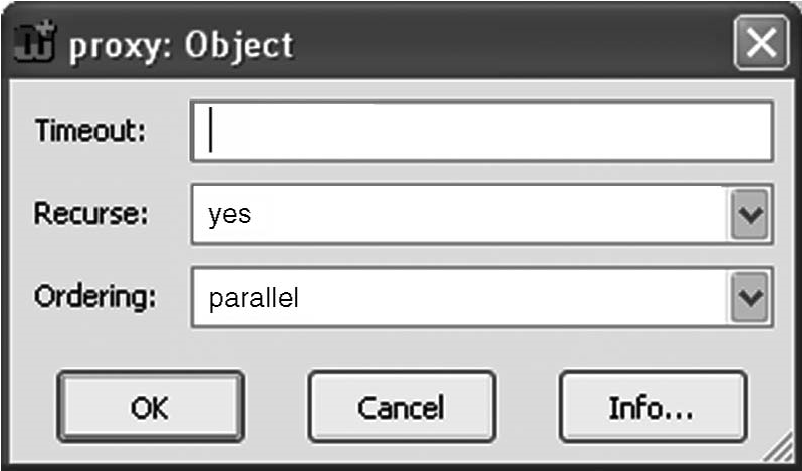


FIGURE 5.2 Specifying a Proxy in CPL using the Proxy language concept

What does the above look like in the modeling language? Figure 5.2 illustrates the use of the Proxy modeling construct when specifying a proxy instance. Having now defined the Proxy concept in the metamodel, all CPL models for specifying proxies follow the DTD. By entering the values based on the language element, a proxy causes the triggering call to be forwarded to the currently specified set of locations. We discuss these connections and constraints with other concepts later.

Other signaling actions, Redirect and Reject, were specified similarly. Redirect has just one property to choose: should the redirection be considered permanent or temporary? Reject states that the call attempt is rejected by the server. The Reject language construct was defined to have two properties: a mandatory status to inform why rejection was made and an optional Reason to describe why the call was rejected. Both Redirect and Reject immediately terminate the call processing execution, so these concepts were defined to have just incoming flows, that is, they form the leaves of the tree.

During language definition, the proxy concept was found to be a suitable place in the language to integrate with another domain, VoiceXML. VoiceXML is used to specify voice messages during call processing. For the operator, VoiceXML was added to the language as its own modeling concept and called by the Proxy via a relationship. This allowed a Proxy to connect the call based on its conditions and, for example, play instructions defined with VoiceXML. The actual voice was already defined elsewhere, and thus the VoiceXML concept just referred to an existing voice specification. Another option considered was integrating metamodels: putting the languages of CPL and VoiceXML together. This was abandoned by the operator as it was seen that different people would create these specifications and there were no real integration needs other than calling VoiceXML.

Switches for Specifying Choices Choices are expressed in CPL with a switch concept. The choice arguments are taken from call requests or from items independent of the call. These are usually entered as property values for the switch. If an entered value is matched during CPL service execution, the choice meeting the condition is chosen. In the case of switches, the condition is simply True or False. The choice results are specified by connecting a switch to another modeling concept using a

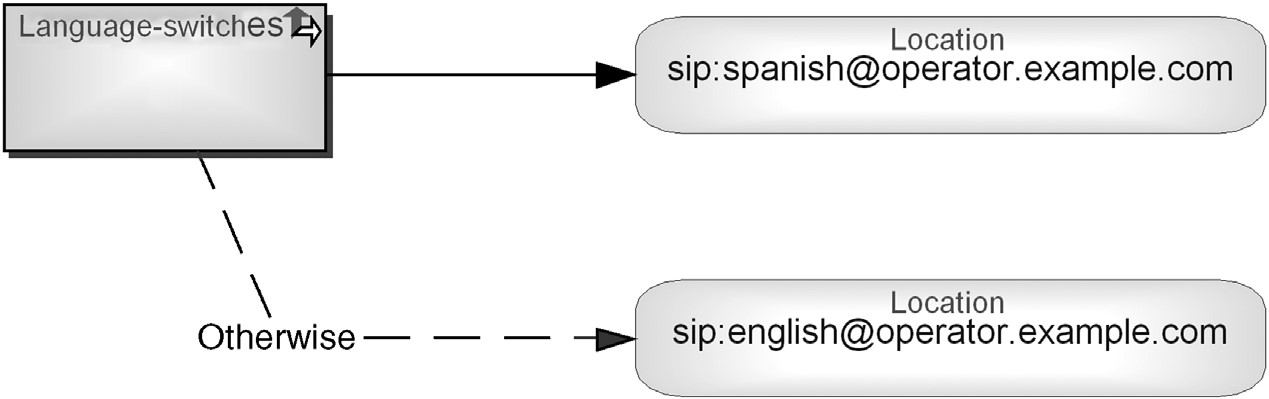


FIGURE 5.3 A specification describing a choice expressed with a switch

relationship. These relationships are divided into two different kinds: a Default relationship, which specifies the result when the condition is met, and Otherwise which specifies the result when the condition is not met. The name of Otherwise was taken directly from the CPL terminology. Figure 5.3 illustrates this: the Default choice is made when the caller wants to communicate in Spanish; otherwise English is chosen.

CPL recognizes different kinds of conditions. These were represented in the languages using different switch types, that is, with dedicated object types as modeling concepts for each switch type. Each switch type thus had different kinds of properties to be entered. Their definition was taken directly from the DTD as with the signaling actions discussed earlier:

. An address switch specifies decisions based on one of the addresses present in the original call request. For this purpose the modeling concept needs two property types: Field and Subfield. The value for Field is mandatory and is implemented as a list of fixed values, where the most common value is already given as a default value. The Subfield is used to specify which part of the address is considered. To support the modelers, a predefined list of possible address parts is provided in the language. Because the Subfield property is optional, no default value was specified. If no subfield is specified, then the entire address was used when evaluating the switch. The results of the address switch were represented along with the Address switch concept: a string value to be matched and a matching policy. An alternative approach would be to show these values by placing them into a relationship pointing to the next element. One reason favoring this choice would be to support reuse of switches as then condition values that depend on the case would not be reused, just the switch condition. This was abandoned because other switches would behave differently during modeling. For example, the Language switch, discussed next, has just one value for comparison and would then be empty.

. A Language switch specifies the communication language the caller wants to use. In the modeling language, this was implemented by adding a property type for entering the preferred language. This property was defined as a string, although predefined values could be added to speed up modeling. If the value entered is matched during call processing, execution of the specified language choice is made. If a match is made (True), a choice specified using a Default relationship path is followed. See Fig. 5.3 for an illustration.

. The string switch is needed to specify decisions based on free-form strings present in a call request. It has one property to give the string value and two additional fields to specify in more detail how the string matching is done and what protocol is used (e.g., who initiates the call). To support capturing these, two property types were added to the String switch modeling concept.

. The time switch is needed to define time- and date-dependent execution information. For example, during holidays calls may be directed differently. In the modeling language, different time and date values were specified as property values of the Time switch. As it was possible to define 19 different kinds of properties, out of which actually very few are used at a time, the timing properties were divided into two categories. Two mandatory values had their own property types and the rest of the timing values were defined using an optional list. The list was defined as pairs of a timing property and its value. The property types were named as in the XML schema for CPL to keep the generator simple.

. The Priority switch allows a CPL script to make decisions based on the priority specified for the original call. This priority switch concept was defined so that the modeler could choose among predefined priorities and conditions. An example of a priority is “less than an emergency.” The value “normal” was defined as the default priority as in CPL, to be used when no priority is given. In this respect, the priority could also have been left empty, or the generator could replace empty values with the default value during the generation. Showing the default value during modeling was considered to make the specification process more transparent and service specifications easier to understand.

Although each switch was described with its own language concept, an alternative possibility considered was to have only one switch concept that would refer to different switch types. Then a modeler would first explicitly choose to add a switch into a model and after that choose which kind of switch is needed. The benefit of this approach would be that the existing switch instances could be changed to new types without deleting the main switch element. This approach is useful when the rules and relationships are the same among the (switch) types. As a single CPL design was considered relatively small, less than 21 main concepts, it was considered more straightforward to use distinct types. This also meant that the switches could be used similarly to other concepts that had been implemented as their own types.

Location Modifiers for Accessing Location Data The behavior of many signaling actions is dependent on the current location set specified. For example, if the user location is Spain, a call can be directed differently than if the location is something else (see Fig. 5.3). The set of locations to which a call can be directed is kept in a CPL server. To modify a location, the modeling language was extended with constructs called location modifiers. They allow adding and removing locations from the location set. Following the XML schema, three different modeling objects were added to the language:

. Explicit Location are set by adding the given URL address to the location set. The location can be further refined with its priority in the location set and with an optional property to choose if the existing location set is cleared before adding a new location. Thus the property specified here is used as an argument to access the CPL server’s service.

. Locations can also be set from external sources. For this purpose, CPL has a location lookup concept. This was represented as its own concept in the language. Location lookup has three properties: Mandatory Source to give a location from which the CPL server can ask for the location, Time-out to specify the time to keep trying before abandoning the lookup, and whether the current location set needs to be cleared before adding the new location.

. A call processing service can also remove locations from the location set. For this purpose, the Location removal concept has a property to specify the location as a string value. If the value is left empty, the default in the concept, all locations are removed from the location set.

The locations were also specified as their own concepts because their connections and rules are different. Subtyping a single location concept, for example, with a property type having different location types as values would not make sense. In this respect, language definition followed the choices already made with switches. The main exception between these choices was that the metamodel did not allow such rich possibilities for choices. For example, Location lookup has the possibility to define three alternative choices, whereas switches allow only two (default and otherwise). Another difference is that lookup, can not be followed by another lookup, whereas switches can be next to one another in the call processing path.

Nonsignaling Actions To record actions that were taken, a server can send mail to the user or log information about the call. In the language, both were defined as their own modeling objects since they had different properties: The Mail concept in the language was defined to include a “mailto” URL property, whereas the Log concept contained information about the log name and its more detailed description.

During language creation, the concepts acting as modeling objects were formalized into a metamodel. The metamodel in Fig. 5.4 illustrates the definition of the object types of the language in the metamodeling language described in Appendix A. All the main domain concepts were defined as subclasses of an abstract Node element to follow the structure of the CPL specification. The object types, however, were defined incrementally, and thus the metamodel shows only the final result.

Model of Computation In addition to the domain concepts presented above, the flow model needed additional modeling concepts. A root node was added to the language to specify a starting point for the call processing service. This modeling concept did not have any properties but just marked the initiation of the call session— making or receiving a call. Although it would be possible to identify the start of the

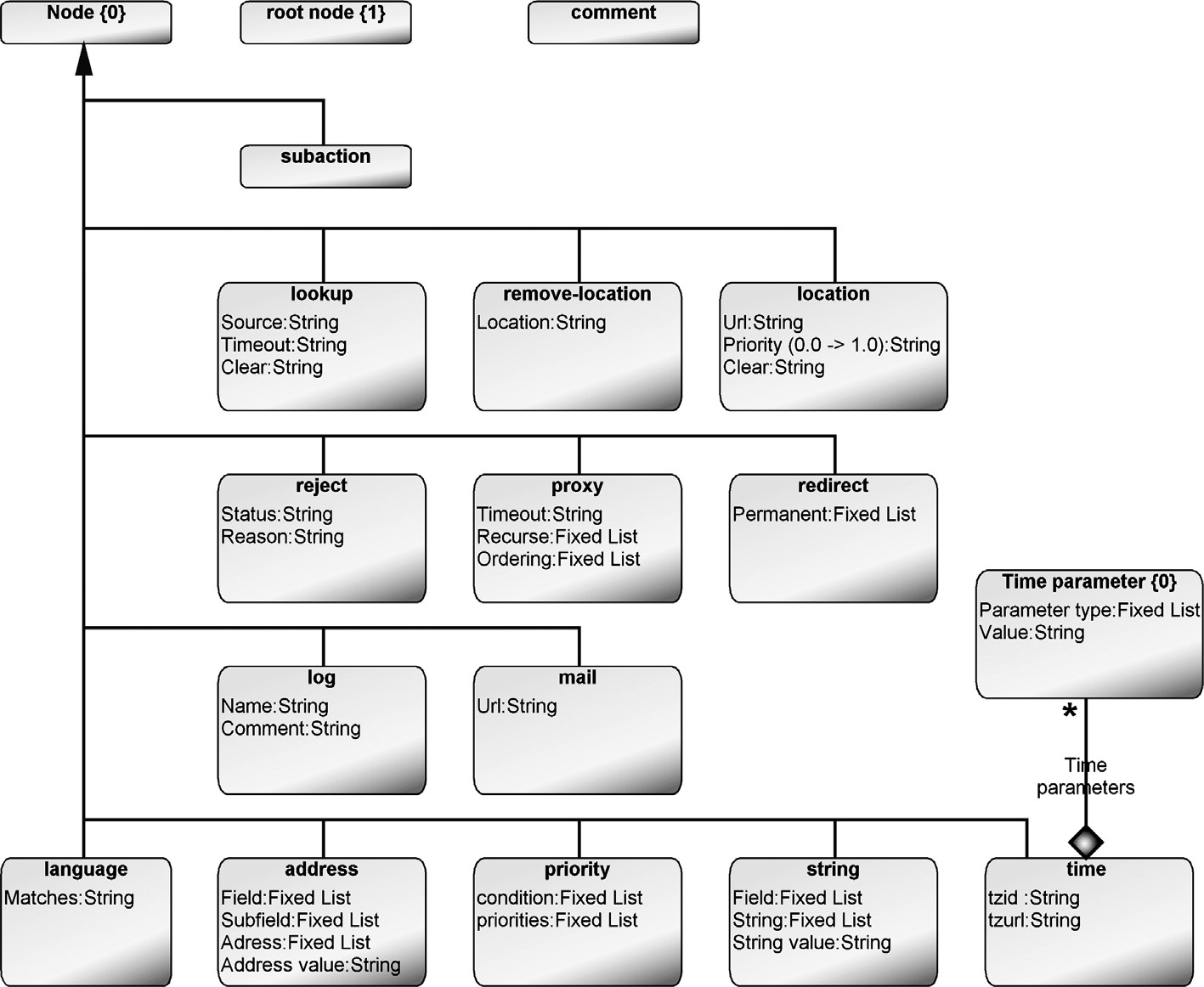


FIGURE 5.4 Metamodel specifying the object types of the language

service from a concept in the model that had no incoming relationships, this policy could easily lead to conflicts if a model is not complete. It was also considered easier to understand the specifications when there was a single starting point.

The start of a call processing service with the Root node was implemented in the language with a Session Phase relationship from the Root node. This relationship was further classified into incoming or outgoing calls. This information is important in CPL for performing any necessary initialization, such as emptying the location set in the server. The modeling language did not include any explicit service end concept. The execution of the service ends when the last element in the flow is reached and performed. At this stage, a CPL server takes the resulting action and the service ends.

The flow of a call was divided into two parts. The main service flow, described using a directed relationship, was called Default path. The name was taken directly from the CPL schema. To model alternative binary choices with the switch concept, an Otherwise relationship was added to the language. Later, dedicated relationships for both Proxy and Lookup were added to extend the default path to specify the predefined kinds of outputs. Default or Otherwise relationships could not be used as they did not have properties to specify outputs. Adding them to the current relationships was abandoned as that would make the language more complex to use: a modeler would be asked about design data that could be unnecessary. To make modeling easier and hide complexity, these choices were put directly into the metamodel.

Model Layering with Subactions CPL identifies two layers to support reuse and modularity. The main action is treated as a topmost action that is triggered, via the Root element, when events arrive at the server. As many services have common functionality, these could be treated as reusable services to be called by other actions. These subactions were defined in the language by enabling an action to decompose into another model treated as a submodel. As the subaction could be defined using the same language concepts as the main CPL definition, there was no need to have a separate language for the subactions. In the metamodel, this was achieved by defining decomposition from a Subaction concept to the Call Processing graph. This language structure also supports reuse well as any Call Processing specification can be used as a subaction elsewhere. Figure 5.5 illustrates possible subaction calls: A design model becomes a subaction when it is called by a subaction in another model. This choice influenced the generator very little: the root element specifying the initiation of the call processing service was just ignored during generation. The model was the same independent of its use as a subaction or main action.

The choice of whether the subaction should be moved to the main specification model or kept in its own separate model depends on the modeling process: if changes in a subaction need to be propagated to all specifications using it, the subaction should be kept in its own model. This capability to support reuse was thought to be a useful solution for service providers and server administrators. The end user was thought to prefer a simpler approach as chances for reuse would not be so common. Also, the end user would be less experienced and having to understand reuse and model hierarchies might require learning things that are rarely needed. To outline the content of the subaction, the language and related tool support were implemented so that the main

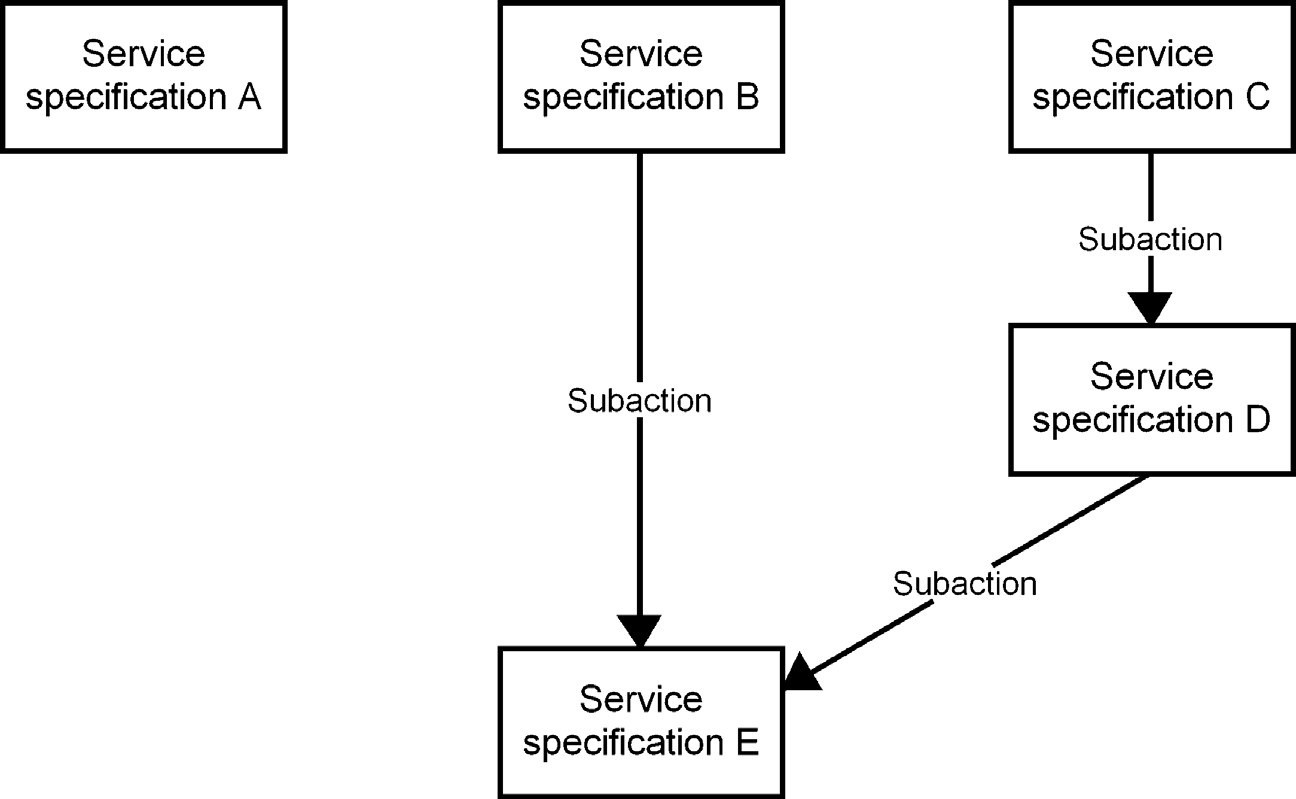


FIGURE 5.5 Reusing a model as a subaction

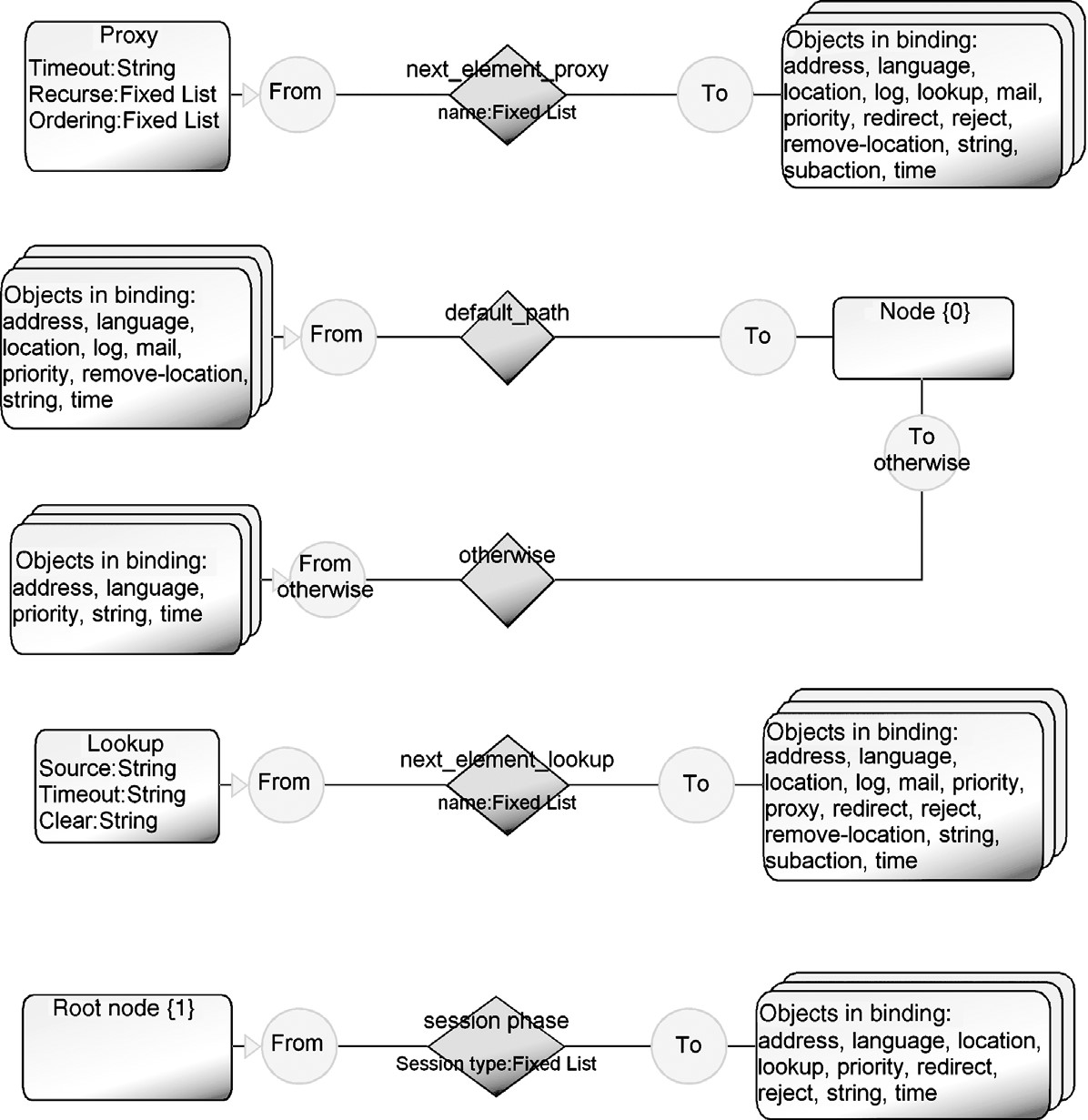


FIGURE 5.6 Metamodel of the language

contents of the subaction were summarized in the main model within the subaction object that referred to a subaction model.

5.3.2 Modeling Rules

Along with identification of the modeling concepts, many of the constraints and model correctness rules were identified. Whenever possible, they were defined to be a part of the metamodel to be checked actively in the course of time. Otherwise they were supported in related model-checking reports. Figure 5.6 gives an overview of the call processing language as implemented in the MetaEdit+ tool. The metamodel shows the modeling objects and their bindings: possible connections via relationship types and role types.

Tree Structure By definition, CPL scripts should never include loops or recursions. This, however, does not mean that models should always form the structure of the script. Model elements should be made reusable, and links to already existing services could be provided to support efficient modeling work. Most constraints ensuring a tree structure were enforced in the modeling language. During code generation, remaining illegal structures for XML (but legal for the model) were identified and forced to follow a tree structure. To support the call process view, the modeling language had the following rules defined:

. A root element could occur only once in a model, that is, only one start element for each service specification was allowed. In the metamodel, the root element further has a constraint that a root can only be in one relationship to start the call process. Reusable subactions could have their own root element, which would be omitted during generation.

. As both Redirect and Reject immediately terminate the call processing execution, these concepts were defined to have just incoming flows; that is, they form leaves of the tree.

. Cyclic structures between the objects were reported as errors.

Choices and Conditions in the Call Path The metamodel was also extended with concepts and rules to enable choices. This was done by having special relationship types for connecting choice types, and related rules called bindings, that specify the legal elements as source and target. The rules included the following:

. The call processing path was initiated with a dedicated relationship (session phase) from the root element, specifying if the call was incoming or outgoing.

. For most choices two alternative path relationships were added, called default and otherwise. The otherwise path was defined to be legal only for selected call elements, namely, address, language, priority, string, and time. See Fig. 5.6 for a detailed metamodel.

. As lookup had three alternative types, their value was added to the dedicated relationship type (called next\_element lookup). This relationship could only be initiated by the lookup element.

. Similarly for the proxy: a relationship with a property type for proxy output was added. This relationship could only be started from a proxy element and had five different output values. Use of the same values multiple times for the same proxy instance gave a warning. Having just a warning was preferred, as it would allow changing output values without first deleting existing values.

Rules for Model Hierarchy To support reuse of other call processing specifications, the language provided a subaction concept. Each subaction object could include only one subaction specification. During code generation, subactions were produced first, followed by the main call specification.

Model Checking Reports In addition to constraints and rules attached to the metamodel, separate model checking reports were also made. These reports analyzed models for the kinds of rules that cannot or should not be checked after each modeling action. For instance, an unreachable part of the call model and a missing default path are best warned about with reports. Similarly, if choices based on a location lookup

### MODELING IP TELEPHONY SERVICES

include the same lookup value multiple times (e.g., more than one success choice), their occurrence can be reported.

Model checking results were accessed in two different ways: separately from a tool menu and by placing the current status and warnings next to a separate model element. A third option would show the warning next to each model element having an error, but combining them in one place was preferred.

5.3.3 Modeling Notation

Lennox et al. (2004) proposes a very basic notation for the language: directed arrows between boxes (see Fig. 5.1). This is a typical starting point when defining a concrete syntax for a language. The notation was made by a consultant. During the notation definition, no customer feedback was given, perhaps because there was no established practice in either of the companies—anything was considered better than writing specifications directly in XML. First, the notation was just boxes but later symbols that clearly distinguished the concepts were used.

The key principle in choosing the notation was that it would be possible to read the whole specification from thevisually represented model. This was a realistic objective because each concept had relatively little data. The notation applied a visualization pattern based on the type of concept: similar kinds of language concepts were represented with the same shape. For example, all switches were represented as rectangles and locations as rounded rectangles. To further improve the readability, small icons were used in symbols to illustrate the concept: all switches have an icon representing choice; for example, the time switch has a small watch icon. Similarly, the relationships used to mark the Default path and Otherwise path were also graphically distinguished: the usual default path, including root start and signaling actions, was represented with a solid line and the Otherwise path was represented with a dotted line. The relationships also showed the path values, such as choices made for proxies and location lookups. Root was illustrated using a circle similar to the start state in many state transition diagrams.

To make the notation aesthetically pleasing, the language was finalized by adding different colors; fill colors for the symbols and icons made the notation easier to read. For example, all signaling actions were represented in light brown. As colors do not work well on monochrome printers, their value in the notation was limited to computer use: inspecting models in the modeling tool or generated documentation.

5.4 MODELING IP TELEPHONY SERVICES

5.4.1 Example Model

The structure of the CPL language maps closely to its behavior so that a service developer who understands the domain concepts can also understand and validate the service from the graphical models. Figure 5.7 illustrates the specification of a

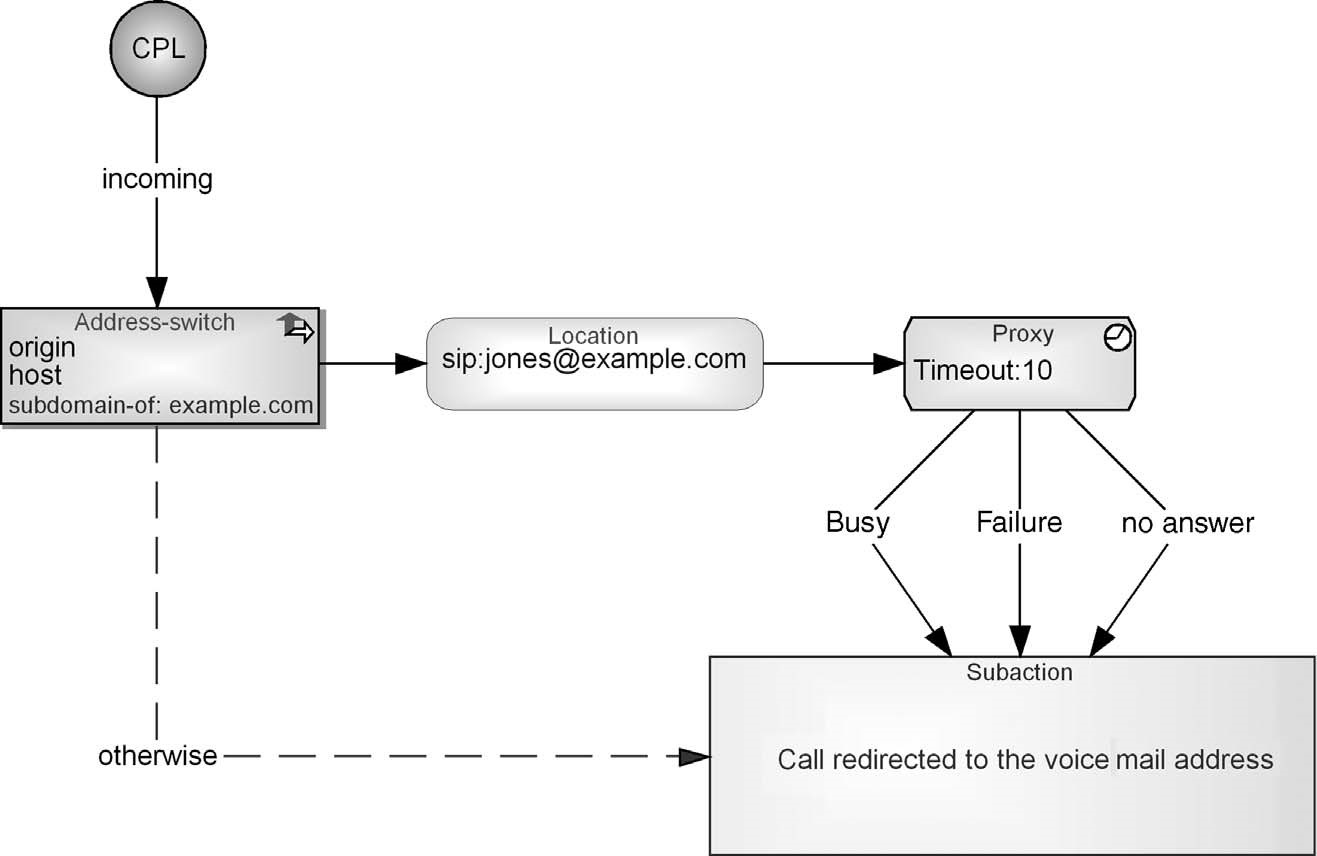


FIGURE 5.7 A sample of call redirecting service expressed in CPL

service: here, all incoming calls originating from “example.com” are redirected to the address “sip: jones@example.com,” and if there is no answer, a line is busy, or a failure occurs, the call is redirected to the voice mail address “sip: jones@voicemail. example.com.” All calls that originate from addresses other than example.com will be redirected immediately to the same voice mail.

5.4.2 Use Scenarios

The DSM solution was made to support the reuse of services as a subaction (see Fig. 5.5, model layering). In Fig. 5.8, the same subaction is reused as in Fig. 5.7. In this case, the user attempts to have his calls reach his desk; if he does not answer within 8 second, calls from his boss are forwarded to his mobile phone, and all other calls are directed to voicemail. If the call setup fails, the failed call information is stored in the log and sent to the email address. This kind of structure was seen to allow providing a library of basic services or their parts to speed up service creation based on existing subactions.

5.5 GENERATOR FOR XML

Defining generators to produce XML is often quite a straightforward process: elements in a model and their connections are described by XML tags. This becomes even easier if the modeling language maps well onto the XML schema. This was true in this case, too, since both are designed to be a good way of describing the same domain. Where XML schemas have had to sacrifice understandability to cope with

### GENERATOR FOR XML

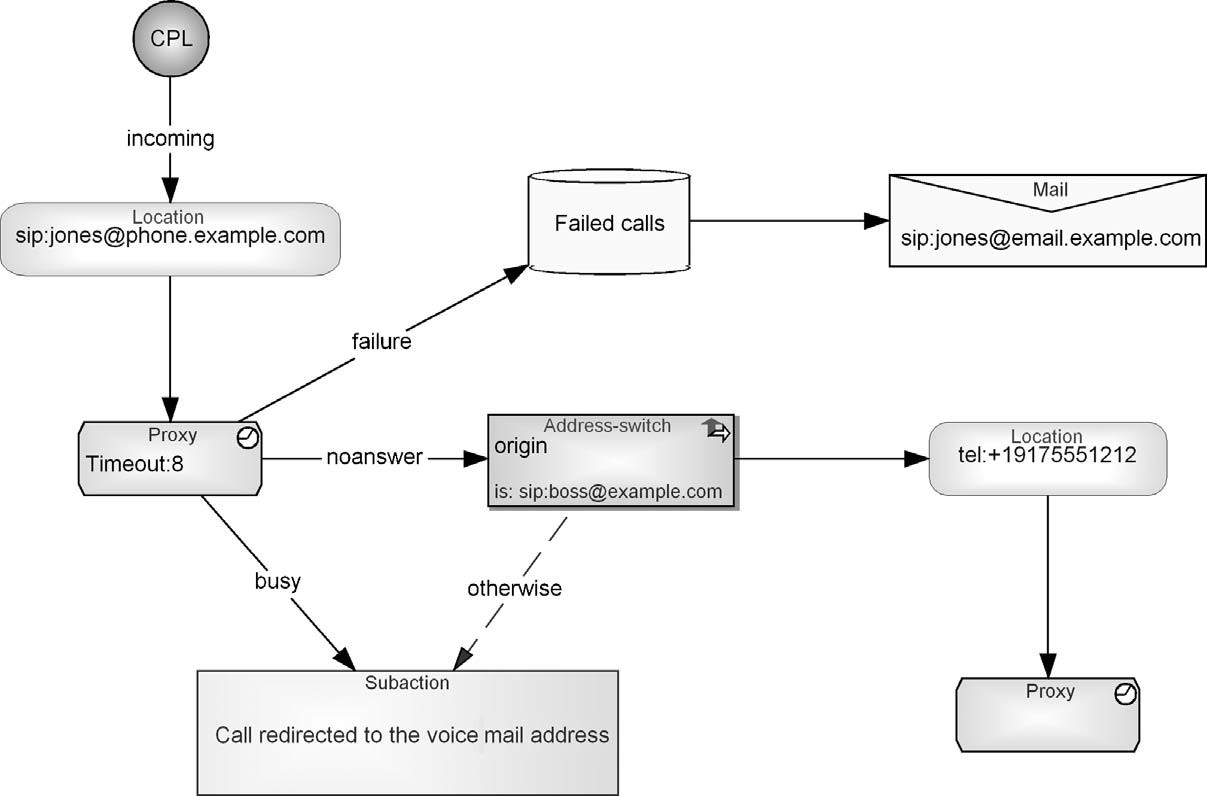


FIGURE 5.8 Call processing service reusing the same voicemail subaction

the limitations of XML, the modeling language can do things in a more natural way. In this case, the generator will do a little extra legwork to produce the verbosity and duplication required in XML.

The first version of the generator was made as a proof of concept for the customer. Later, the generator was completed based on customer feedback. More effort was needed for defining the default behavior for different types and defining the XML tags for the ending elements correctly. The generator was defined in parallel with the modeling language: when a new modeling concept was added to the language, a generator module related to the added concept was implemented too. This approach allowed testing the language using the example specifications given in the original language specification. Further testing of the generated result was done using basic XML schema validation tools, as there were no actual CPL engines available.

From the designs expressed in the models, the generator produces code that can be parsed and then executed in the CPL server. Let’s look next at the structure of the code generator and then samples of the generated call processing scripts.

5.5.1 Generator Structure

The generator visits each element in the model, calling a generator module for that element’s type (see the visitor pattern in Chapter 11). Figure 5.9 illustrates this generator structure by describing the generator modules and call structures between them. The generator modules that handle the CPL concepts are emphasized with a gray rectangle. Most of the other generator modules provide generator functionality that was common and applied at many places during the generation.

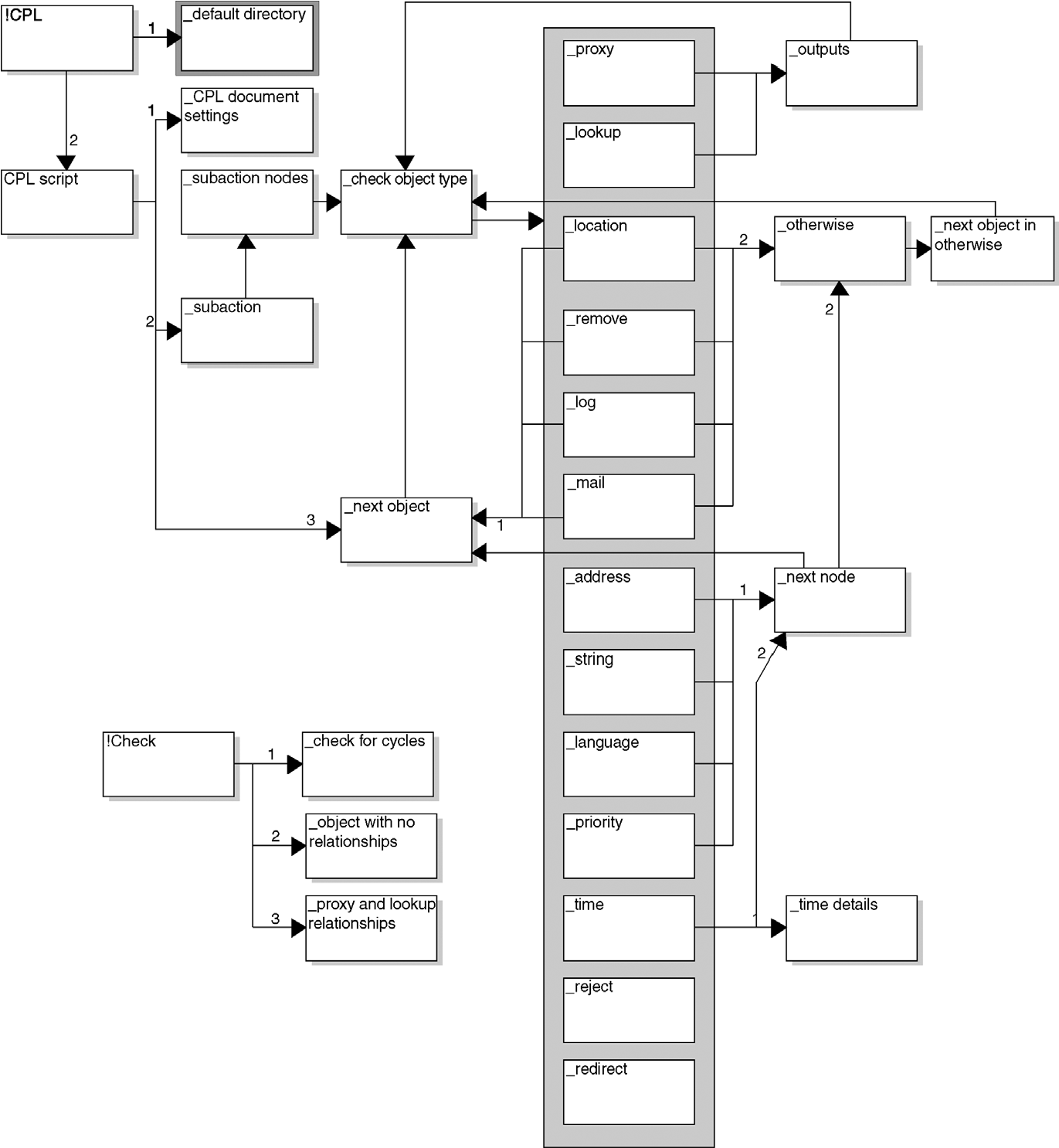


FIGURE 5.9 Structure of the CPL generator

Generators were defined for three different purposes: producing the service specification into a file, producing it into an output window, and checking the service specifications. In the figure, these main generators are represented on the left-hand side. The difference between “!CPL” and “CPL script” is minimal as the only difference is how the generated result is shown: saved into a file in a default directory or shown in the generator output editor. Therefore, these two generators use the same generator modules.

After having defined the document settings (XML header), the generator looks for subactions to be generated. This was needed because CPL expects all subactions to be defined in the beginning of the service specification. This is followed by choosing the domain concepts to be generated: the “\_check object type” module chooses the domain concept found in the call processing specification and runs the right domainrelated generator module (inside the gray box in Fig. 5.9).

### GENERATOR FOR XML

So each concept had its own generator module. As some concepts resembled each other (e.g., all switches), a single generator module could serve more than one concept. This was mainly done because the generated outcome looked the same for all modules. Generator modules named after domain concepts then called other generator modules—mostly those applied to navigate further in the specification, such as the generators “\_next object” and “\_next node.” Embodying common behavior, they were thus defined to be reusable.

The naming of modeling concepts was taken directly from XML to make generators easy to make: a generator uses domain concept names while accessing the model and producing the required output. As XML required a special naming policy, domain concept names using a capital letter were changed to lower case. Generating XML is problematic for reserved words and white spaces in entries. For the former, regular expressions were used to describe legal text strings and for the latter a generator was defined to make entry values legal in XML (e.g., remove spaces from domain concept names).

To produce the conditions in the same order, the generator first produces the default path information followed by the “otherwise” path. The generator implements this by using the metamodel data—access first the default path and then the otherwise path. For CPL this was an adequate and simple solution. If different path execution orders had needed to be specified, then the path choice could have been given to the relationship as a property describing the path. Giving a number to the relationship path is one approach. If there are only two paths from the switch, or if it is relevant to describe the execution order only for one of the many paths, such as default, first, and so on, marking just one path is enough. The path order could also be taken from the order of specification—the path specified first in the model is executed first—or from spatial information—the path specified highest is taken first. These choices, however, could restrict the modeling process, and reuse of larger model chunks between different specifications would not necessarily work—the context would be different between models in a library and their use in design.

As not all design rules can be defined in the modeling language to be checked at modeling time, model checking generators were defined. Rather than calling individual model checks, they were all called by one checking generator named “!Check.” This generator checked a service specification and created warning reports about cyclic specifications. The result of the warning report was defined to be shown in a separate window, although an other possibility would have been to place the warning information as a visible part of the designs. One approach considered was showing in the model that there were inconsistencies, but not showing the details; the details could then be seen by running the complete model checking report.

5.5.2 Generator in Action

The task of the generator is to follow the nodes via their connections and create the corresponding CPL document structure in XML. The listing below shows an example of the generator output. The XML is produced with the generator from the service specification illustrated in Fig. 5.7.

Listing 5.1 Call redirecting based on location of caller origin.

1. <?xml version="1.0" encoding="UTF-8"?> 02 <!-- DOCTYPE call SYSTEM "cpl.dtd" -->
2. <!-- DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL

1.0//EN" "cpl.dtd" -->

1. <cpl>
2. <subaction id="voicemail">
3. <location url="sip:jones@voicemail.example.com">
4. <redirect />
5. </location>
6. </subaction>
7. <incoming>
8. <address-switch field="origin" subfield="host"> 11 <address subdomain-of="example.com">
9. <location url="sip:jones@example.com">
10. <proxy timeout="10">
11. <busy><sub ref="voicemail" /></busy>
12. <noanswer><sub ref="voicemail" /></noanswer>
13. <failure><sub ref="voicemail" /></failure>
14. </proxy>
15. </location>
16. </address>
17. <otherwise>
18. <sub ref="voicemail" />
19. </otherwise>
20. </address-switch>
21. </incoming>
22. </cpl>

The generator starts by going through all the subactions of the service specification. This example contains only one subaction, the voicemail box at the bottom right corner of the model, for which the generator produces lines 4–8.

1. <subaction id="voicemail">
2. <location url="sip:jones@voicemail.example.com">
3. <redirect />
4. </location>
5. </subaction>

This “voice mail” subaction defines a location element (line 5) as well as a redirect element (line 6), which activates that new redirection automatically.

After producing subactions, the generator starts to specify the main call processing actions. It goes through the service specification from a service start (the “CPL” circle in Fig. 5.7). The generator crawls the connections from the CPL circle through the “Incoming” relationship to the Address-switch object. It produces the properties of the Address-switch node as element attributes in the generated output (lines 10–11).

1. <address-switch field="origin" subfield="host">
2. <address subdomain-of="example.com">

The generator continues to follow the main flow path arrow to the next object and produces the location definition (line 12).

### FRAMEWORK SUPPORT

1. <location url="sip:jones@example.com">

The path continues and the proxy handling is generated on lines 13–17: first the timeout attribute (line 13), followed by three alternate connections from the proxy element.

1. <proxy timeout="10">
2. <busy><sub ref="voicemail" /></busy>
3. <noanswer><sub ref="voicemail" /></noanswer>
4. <failure><sub ref="voicemail" /></failure>
5. </proxy>

Finally, the generator produces lines 20–22 for the cases where the call origin has an address other than example.com.

1. <otherwise>
2. <sub ref="voicemail" />
3. </otherwise>

The generated code forms a complete service whose validity has already been checked at the design stage. Because the modeling language contains the rules of the domain, service creators can only make valid and well-formed design models. The modeling language can also help service creators with consistency checks and guidelines, for example, by informing them about missing information (such as no call redirect being specified). These rules are highly domain-specific and thus can be handled only with a domain-specific language.

The generator producing the call processing scripts, and related model checking, was tested against several test cases. First, script samples were taken from the CPL specification, but further cases were made to cover situations that were not available in the specification documentation. In total, about 40 different kinds of service specifications were used and found to be enough to test the possible combinations. Then the generator was executed and the result was compared first for legality as XML and then against the test cases.

* 1. FRAMEWORK SUPPORT

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 largely defines what the generated code should look like. Similarly, for the VoiceXML extension for the operator, just a simple link to external files was enough. If the DSM were to be extended to also cover VoiceXML, most likely no framework code would be needed there either.

The generated code, the call processing service specification, can thus be immediately executed in the CPL server. This target execution environment provides default behavior for situations where the CPL specification is left open and for cases where a failure may occur. For example, if locations are not found for the call target, the failure is handled by the server. Here, service execution would be terminated and the default behavior is provided. In this sense, CPL already provides a good target platform for specifying services.

For the equipment manufacturer, the target configuration and generated Java would obviously need some framework code, but these will not be discussed here. We discuss Java code generation in two other chapters with different framework support. We show first in Chapter 6 how static structures are generated into Java code and then in Chapter 9 how framework support can be applied to code generation that also covers the behavioral side.

* 1. MAIN RESULTS

The DSM solution for CPL allows call processing services to be specified graphically. Services can be created with a language that almost solely applies call processing concepts—concepts that every service engineer must know anyway. A code generator reads the created service specifications to produce service specifications in XML. The DSM approach provided the following benefits:

. Service creation became significantly faster. Making small services with the created language and comparing it to earlier manual practices showed that DSM lead to processes that are 6 times faster. It was further expected that when the specification size and complexity become larger and subactions can be reused in the model levels, the difference would be even bigger. This is natural as XML is not really made for reuse.

. Service creators do not need to master XML. With the created DSM solution, service engineers need not deal with XML concepts at all. XML was just used internally as an intermediate output for passing call processing specifications to the CPL servers.

. Specifications could be checked during modeling. Checking the correctness of the specifications was valued highly, especially because call processing services for IP telephony were new. The simple statement that “if the specification can be drawn, it will execute in a server” was very powerful and easy to understand by personnel. Model checking was mostly done at modeling time, but some checking by generators was also found useful.

. The quality of specifications was better. The first tests showed significant decreases in errors and using generators totally removed most typical errors.

5.8 SUMMARY

This first case of DSM creation shows how a language is created based on a wellbounded domain specified as an XML schema. Schema concepts are mapped almost one-to-one to language concepts. Thus the modeling language concepts are directly

### SUMMARY

based on the key elements and the structure described in the CPL specification. Language elements such as Switches, Locations, and Signaling actions and their attributes are specific for processing Internet phone calls and services. These concepts automatically become familiar to the service developer and they form a comprehensible working environment for CPL service developers.

The modeling language was built before any CPL servers were implemented, so the platform was not yet ready. However, the CPL DTD and later the companyspecific extensions allowed building the DSM solution in advance. Adequate specifications of the language enabled this parallel development of the DSM and of the platform. This case shows that there is no need to wait until the target environment and platform are ready: DSM can be made ready beforehand. This can be useful when service specifications are needed to be made before they can be applied.

On the code generation side this case was very clear-cut. This was mainly because the language concepts are already defined as elements in XML, and the property values of the modeling constructs are attributes of the XML elements. CPL servers already provided an interface specification and thus showed the expected input.