

Chapter 5

Interaction

Guided interaction is based on a conversational or directed dialogue model [90]. A client initiates a dialogue by requesting a service provider to perform a capability. A *guide* is a type of mediator or facade which presents a “user friendly” interface to a back end service. The guide is responsible for telling clients what the service can do and once a client has determined that a service can provide the required capability, the guide is responsible for collecting the information the service needs to perform its function. Clients may be other services, software agents or people. The guide or service provider incrementally collects a set of parameter values from the client for submission to a back end process.

The purpose of guided interaction is to facilitate communication between two entities that have little or no prior knowledge of one another. It is assumed that the client has an independent goal and has discovered a service (via some discovery mechanism) that can assist it with the achievement of its goal. While the client does not have to know the exact names, datatypes and order of the parameters beforehand, the client does have to have access to some form of appropriate data before engaging with the service. For example, before engaging a bank service the client will have information that is relevant to banking such as account numbers and transaction amounts.

Guided interaction is designed to enable interaction between software entities that have not been explicitly pre-programmed to interact with one another. It can not achieve this goal and at the same time display optimal computing efficiency. Clients with prior knowledge or experience with a guided service may not need to use the mechanism described in this chapter for repeated interactions.

Guided interaction uses conversational dialogue to allow heterogeneous services to exchange information with one another at runtime without prior agreements in place regarding the syntax and semantics of the data they will share. The guide cannot completely remove the complexity involved in interacting with unknown partners but it does introduce that complexity incrementally.

If a client does not understand a particular request for information from the service it can ask for help or alternative information. This means the client does

not have to know the exact form of the service's operation signatures, or the correct order of invocation of operations in advance as this is taken care of by the guide.

There are two advantages of a dialogue approach to dynamic service interaction. The first advantage is that clients do not have to know the service's interface in advance or interpret a WSDL document on-the-fly. The second advantage is that clients using a conversational mechanism can interact at runtime with *any* service that provides a guide rather than being tied to specific service implementations via hard coded calls to WSDL interfaces.

There are several ways guides could be implemented and used: a guide could be provided as an alternative interface to a WSDL service. A service provider could make available a guide that facilitates access to several of its services including those described in WSDL specifications. Another alternative is that independent third party providers could implement guides for any services that have publicly accessible interfaces.

The promise of ubiquitous services means that pre-programmed one on one interactions will not have the flexibility required for dynamic interaction with newly discovered services. A means of interaction that is flexible and robust with a clear achievable semantics is needed to advance the vision of ad hoc interaction. An asynchronous directed dialogue mechanism [166] for one to one service communication is necessary to enable ad hoc interaction between the many diverse kinds of services that will be developed in the coming years.

The rest of the chapter is structured as follows. The next section (section 5.1) draws out some criteria to be used to evaluate solutions to the problem of ad hoc interaction between services operating without human intervention at runtime. The foundations section (section 5.2) introduces several technologies that have influenced the definition of the guide. Section 5.3 presents the details of the interaction mechanism and a demonstration of how it works in the prototype implementation is given in section 5.4. Section 5.5 discusses other work in the area of service interaction languages. The chapter concludes with an evaluation and discussion of the proposals in section 5.6.

5.1 Criteria

The introduction outlined the problems services encounter when interacting in a heterogeneous environment. Here are some criteria that can be used to evaluate solutions to the interaction problem.

A solution should be *easy to use*. The developer of a guide should not have to expend too much time and effort to create interaction specifications. Solutions that require a large effort may not be widely used and may also be hard to verify. The solution should provide well understood primitives that have a broad basis of support and acceptance; such as the "Get" and "Set" methods of Object Oriented programming [21] and the REST [59] operations "Get", "Put" and "Post". Ease of

use does not imply simplistic, the solution must be *expressive* enough to describe interaction scenarios for all types of service capabilities.

A solution should be *unambiguous* it should have explicit semantics especially for a shared interaction language. Both partners in an interaction should be able to understand their options at any point in the interaction process. The solution must respect the independence of both parties by allowing them to behave autonomously within a *loosely coupled* conversation context. For example, both parties must have the option to refuse requests for information or functionality. Another aspect of independence is that the solution should only impose requirements on the behaviour of participants that are related to the provision of the requested functionality.

A solution must be *executable*. In order to demonstrate that the language is sufficient for the purpose and complete enough to describe interactions in the domain of interest, the solution must be able to be implemented, preferably without recourse to proprietary technologies. An implementation which allows both participants to communicate meaningfully in an interaction defined using the provided solution demonstrates that it is feasible and executable.

The solution should allow the description of *flexible* and *robust* interaction plans. Flexibility is required to cater for differing client information resources and competencies. A robust interaction specification would ensure successful completion or runtime *error handling* that delivers specific information about the cause of failure. In the heterogeneous environment that will characterise services in the future, the provision of a form of context sensitive *help* using semantic disambiguation to resolve terminology differences, before a failure is declared, is necessary.

5.2 Foundations

Several mechanisms, paradigms and technologies have contributed to the work presented in this chapter. In exploring the notion of conversational services the answers to several questions were sought, including: What kind of information do computer programs exchange? Can software communicate using a means other than published interfaces? How do programs assist their human users? Who is in control of the interaction? This section describes the technologies that have given answers to these questions. They include Human Computer Interaction (HCI), Application Program Interfaces (API's), Wizards, Intelligent Agents, Protocols, Dialogue systems, Linguistics, XML and Voice XML. A brief introduction to each technology is given and outline which ideas were taken for application in the context of conversational service interaction.

5.2.1 Computer interaction mechanisms

In the early days of personal computers human computer interaction involved using the command line interface [134]. To make a computer do something users needed to enter the exact name of an executable program (command) and its parameters in

the correct order at the command prompt. Although this is a powerful and efficient interface, it means users have to know the commands and the sequence of parameters in advance as parameters are accessed by position not by name.

As programs became more sophisticated it was not always sufficient to have a fixed set of parameters known at startup, it was sometimes necessary to get *input* from the user at runtime. Programs could print a prompt on the screen to tell what kind of data was required and users would enter appropriate data during processing. Alternatively the user would be offered a list of values from which they could *pick* an appropriate value. To reduce the need to know program commands in advance the user could be asked to *select* from a menu of items representing functions the system could perform.

Data entry programs process massive amounts of repetitive input data. Data entry forms were developed to speed the data entry process. Instead of a sequence of individual prompts, the prompt was given next to a text box on a paper form and a trained operator entered the data from the form into the equivalent box on a computer screen. The primary advantage of the form interface is that it facilitates speedy repetitive data entry.

The Windows operating system greatly improved access to programs for most users with its point and click interface, however it did not introduce any new mechanisms for gathering the input from the user. The Windows interface is still based on three input mechanisms: input of a single data item, pick from a list of values and select a command representing an action to be performed. Windows has of course made the presentation of these mechanisms very sophisticated and varied, prompts for input items are seen in such elements as dialogue boxes and forms. Drop down menus and list boxes are also a fundamental part of the windows operating environment.

Application Program Interfaces (APIs) are the program to program version of the command line interface. The API represents what a programmer needs to know in order to use a program [118]. A well documented API contains a description of each operation with its parameters and their datatypes and the datatypes of the values it returns. Ideally the API should include any pre and post conditions that constrain the operations. There are several other requirements for well documented interfaces that were described in section 2.1.

In summary, what is learned from human computer interaction is that initially there were three kinds of programs. The first type takes all its input data as parameters when it is called, it processes this data and returns a result. This mechanism assumes that users know the command name and parameter order before calling the program, it is the precursor of the API and it is the mechanism currently used for many services.

The second type of program uses forms to enable fast accurate data entry. This is in contrast to how forms are used as an input mechanism for services. Services use forms to exchange several parameters at a time within one message. The structure and contents of the form are described in an XML schema and users need to understand the schema before data values can be extracted or inserted correctly.

The third kind of program can interact with the user to get more information during the processing phase. There are three ways programs can get the information they require. They can ask the user to input a value representing a single data item, or ask them to pick from a list of values, or ask for a selection from a list of capabilities. These three types of information gathering are at the heart of the mechanism proposed in this chapter.

5.2.2 Linguistics

In linguistics four types of sentence, declarative, interrogative, imperative and exclamatory¹ are recognised. Declarative and exclamatory sentences declare facts, interrogative sentences ask questions and imperative sentences give commands. Usually only three (declarative, interrogative and imperative) are used in a technical context².

In computer interaction, the purpose of an interrogative message is to ask for information e.g. “get the current time” while the purpose of an imperative message is to ask for an action to be performed e.g. “set the current time to 21-20-33.00”. The purpose of a declarative message is to inform the recipient of information, either the answer to a request for information (“the current time is 22-00-13.04”), or the result of performing an action (“time set to 21-20-33.00”).

The concept of three types of messages accords with the interaction mechanisms looked at in section 5.2.1 i.e. ask for input information, ask for an action by selecting from a menu of commands, and tell answers or results.

5.2.3 Intelligent Agents

Intelligent agents are proactive, reactive, autonomous goal seeking, communicative, and possibly mobile software entities [67, 25]. They use Agent Communication Languages (ACLs) to “talk” to one another. There are two primary agent communication languages FIPA ACL [65] and KQML [121, 102].

KQML performatives and FIPA ACL Communicative Acts (CAs) are derived from speech acts [138]. The term performatives is used to represent both KQML performatives and FIPA CAs. Performatives indicate the type or purpose of a message, such as a query, a response or an action request. KQML and FIPA ACL also provide performatives for networking or group communication and advertising capabilities to other agents.

Both KQML and FIPA ACL have formal semantics for each of their performatives. A good introduction and comparison of KQML and FIPA ACL including their semantics can be found in [101].

The formal semantics of performatives are based on “truth telling” agents holding beliefs, desires and intentions (BDI). The ascription of beliefs, desires and intentions to software entities is well established in AI. However, this is not necessary for

¹www.uottawa.ca/academic/arts/writcent/hypergrammar/sntpurps.html

²mit.imoat.net/handbook/s-types.htm

services and Labrou in [101] reports that many people who develop agent systems view the formal semantics of the performatives as less important than their intuitive meaning.

Performatives form the basis of *messages* that contain three types of information.

1. Information about the context of the message including the name of the sender and receiver, and a reference to the language or ontology used for the content.
2. The performative.
3. The actual content that relates to the performative.

The main idea taken from agents is that it is possible for software entities to communicate using a “conversation-like” exchange rather than a more typical “operations” on interfaces type of interaction. Some other elements of agent communication languages that are useful are: the structured format for messages, and the idea that content of the message is largely independent of the dialogue mechanism.

In regards to specific KQML and FIPA performatives, there are many that are not relevant in the context of one to one service interaction such as those for facilitating multiparty conversations and networking. There is some ambiguity surrounding the “subscribe” and “unsubscribe” performatives as they can be viewed as related to the request for and provision of a notification service rather than indicating the purpose of a particular message.

If the formal (BDI) semantics of the performatives are disregarded there are several that are relevant, for example KQML *Ask* and *Tell* and FIPA ACL *Query*, *QueryIf*, *QueryRef* and *Inform* in relation to information and *Achieve*, *Request* and *Propose* in relation to actions.

A recent discussion³ highlights the similarities and differences between agent and service-oriented applications, with the difference between them being primarily related to their degree of autonomy. Some of the participants suggest there is a convergence between agents, services and semantic services. The interaction mechanism presented in section 5.3 will further assist this convergence by providing a means of communication that bridges the gap between the agent and service interaction mechanisms that exist at this time.

5.2.4 Protocols

Performatives and the set of appropriate responses to them can be seen as the building blocks for protocols. Burmeister et.al. [29] use three basic “building block” performatives *inform*, *query* and *command* to build protocols of interaction.

The responses that can be made to the query and command performatives are variants of the inform message: *answer* in response to a query and *report* or *reject* in response to a command. The sender of a *query* performative is requesting information from the receiver, the receiver’s response will be to tell an *answer* to the sender. The sender of a *command* performative is requesting the receiver to perform some

³See sharon.cselt.it/projects/jade/jade-develop-archive/0321.html

action. The receiver of the command can make one of two possible responses; the first response is to *report* the results of performing the action, and the alternative response is to *reject* the request.

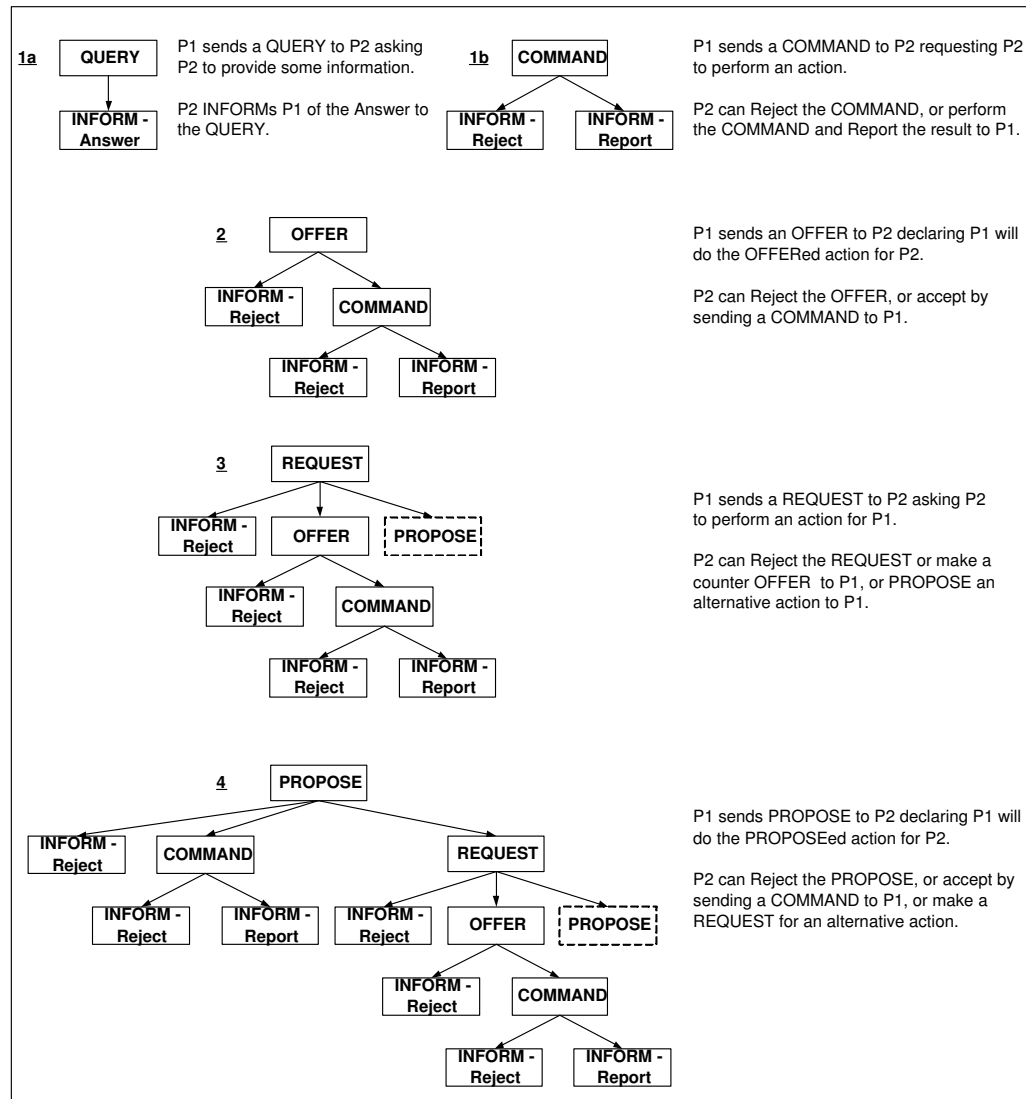


Figure 5.1: Protocol building blocks

Complex protocols can be built from these basic building blocks and figure 5.1 gives an example of how this is done. Two basic protocols are shown at the top of the figure (1a and 1b), with the figures 2,3 and 4 being new protocols defined in terms of these two basic protocols.

In this way, protocols are represented as trees, with each enactment of the protocol taking some path through the tree. An advantage of a tree structure or directed acyclic graph (DAG) for protocols is that the next state is always determined by taking one of the paths from the current state and the history of the interaction is

the path that leads from the initial state to the current state. This allows protocol tracing and debugging to determine the reason for failures or protocol improvement.

Two of the ideas from [29] incorporated into the interaction mechanism are the use of the two basic building blocks query and command because they are similar to the input and pick or select abstractions seen in section 5.2.1. The second idea is that a protocol or performative can be defined in terms of the responses that can be expected to flow from it.

5.2.5 Wizards

Wizards are primarily Windows based programs that help users achieve goals. They do this by guiding users through complex tasks, or tasks that may have many steps and require those steps be taken in a prescribed sequence [20]. Users are guided through the data entry process in the correct order to ensure their goal is achieved successfully. Wizards are particularly useful for novice users who lack the necessary knowledge to perform the task.

The main ideas drawn from this mechanism are that service clients can be viewed as as equivalent to novice human users. Wizards demonstrate that complexity can be broken down into a simpler sequence of interactive steps. This should be done by the entity that understands that complexity best - the service provider.

A service that ensures its users are guided through the performance of complex operations will make it possible for clients to access the service, even when its exact data requirements and interaction sequences are not known in advance. In fact, clients who can allow themselves to be assisted through complex tasks are able to interact with a greater range of services than those clients that specialise in interacting with a single complex service.

5.2.6 Human Computer Voice Dialogue systems

Interactive voice response (IVRs) or dialogue systems provide an interface between human users and computer systems. They use recorded human voices or computer generated voices, to speak instructions and offer options to human users. Dialogue systems are an application of the facade pattern [70] in which the dialogue mechanism stands between the user and the back end applications in order to simplify the system from the user's perspective.

Dialogue architectures are structured around the performance of three functions: The first is to *interpret* (spoken) user input, the second is to *manage* the conversation and the final function is to *generate* output for the user.

The interpretation of human voice input is an active area of research, but in the context of computer to computer service interaction it is not of interest to us. The management of conversations with human participants requires such activities as confirming mutual understanding, making acknowledgements, handling interruptions, providing assistance when required, and managing dialogue obligations such as the obligation to reply to user requests [4].

Allen et.al. [3] hypothesise that within their domain of interest, which is free form human computer dialogue, most of the complexity of interpretation and dialogue management is independent of the specific task being performed. This of interest, because as long as a service (provider or client) has the ability to manage its role within a dialogue system, then it can participate in any dialogue regardless of the actual content of specific conversations.

5.2.7 XML, XML schema

XML⁴ is a text based means of describing machine processable data. XML is based on a subset of SGML (ISO 8879)⁵. XML documents contain elements represented by named start and end tags which contain data. The type of the data is described by the tag name. XML schemas⁶ provide a way of describing the structure, content (elements) and semantics of XML documents.

XML is now a well known technology for the representation and exchange of data across homogeneous and heterogeneous applications. The primary advantage of XML from our perspective, is that it frees applications from prescribed input sequences. XML input allows applications to determine the identity or type of information from the XML tag rather than its position in a sequence such as in an API signature.

5.2.8 Voice XML

VoiceXML⁷ (VXML) is a specification that allows the description of human computer voice dialogues in XML for processing by a VXML engine. VXML is a means to bridge the gap between human users and computers by collecting information from a human user for submission to back end applications. It is based on documents that describe the information to be gathered in the dialogue. VXML uses two primary abstractions for collecting information, forms and menus.

Forms are collections of one or more data items. Each item has an associated “prompt” telling what kind of information is required. When the item is activated (usually in document order) the prompt is spoken to the user. An item may also contain a grammar which enumerates the acceptable values the user can utter in response to the prompt. When all the required items in a form are collected from the user the set of responses is submitted for processing.

The other means of eliciting information from users is a menu. Menus are used to represent both pick lists of values and lists of selectable actions. A list of alternative values is spoken to the user and they select one of these as their response. Menus can also have associated grammars, so an application can accept variations of the

⁴www.w3.org/XML/

⁵www.iso.ch/cate/d16387.html

⁶www.w3.org/XML/Schema

⁷www.w3.org/TR/voicexml20/#dm1AFIA

actual terms used in the prompt, for example if the prompt is “blue shoes” then the grammar may allow responses such as “blue”, “shoes” or “blue shoes”.

The form, as a container for data items and menus which offer lists of alternatives to be elicited from the human user, are abstractions that are familiar from section 5.2.1. The structured specification of dialogues in XML for processing by the VXML engine or interpreter is a useful implementation technique.

5.3 The guide: a dialogue mechanism for services

Guided interaction has two parts. The first part, introduced in section 5.3.1 is a shared language for the exchange of information between services. The second part, introduced in section 5.3.2 is a language and mechanism for constructing interaction plans that are used to generate and interpret messages using the information exchange language.

The shared language allows a guide, representing a service provider, to tell its clients what its capabilities are, and to proactively seek the input data it needs to deliver the capabilities. The language also allows clients to seek further information from the provider to ensure mutual understanding of the requirements.

The mechanism frees the client from having to know in advance the specific names and types of parameters, and the order of operations necessary to access the capability. Clients using guided interaction can interact with *any* guide enabled service at runtime rather than being tied to specific service implementations via hard-coded calls.

The purpose of the guide is to collect a set of parameter values from the client for submission to a back end process. The focus is not on specifying the order in which messages are exchanged, but on specifying the information that is (still) required before the back end process can be invoked. Instead of specifying a sequence of messages, a set of items or parameters is specified for which values are required. The collection of an acceptable set of parameter values directs the path taken by the guide.

A guide has been implemented as two communicating Coloured Petri Nets (CPN)⁸. CPN uses the CPM-ML language for declarations and net inscriptions. A demonstration of how these nets interact using the shared language and a set of instructions is given in section 5.4.

In section 5.2.6 it was noted how some dialogue architectures are structured around the performance of three functions: interpretation of user input, conversation management and generation of output for the user. This division of responsibility is largely mirrored in the guide but the complexity of the interpretation function is reduced by the constraints on the conversation language.

⁸wiki.daimi.au.dk/cpntools/cpntools.wiki

Two scenarios are outlined below which follow from the futuristic scenario presented in section 1.7. These scenarios serve to illustrate how the parts of the mechanism are structured and how they work together to provide ad hoc interaction between services.

In the first scenario, the client (Martha's PDA) has been directed to a service provider (currency conversion service). The PDA/client is unsure of the exact terms the provider uses to represent its capabilities. The client asks the provider to do the generic capability "Menu" to find out what the provider can do. The provider however, restricts its capabilities to those who register or login, so the first response from the provider requests the client to select one of "Register", "Login" or "Exit". The client matches the menu items with its internal list of goals and elects to login. The provider assists the client through the login process by requesting the specific information items it requires. After the login process is completed the provider offers the client a menu of the capabilities it can deliver.

In the second scenario, the PDA (client) has been offered a list of capabilities and it has selected "ConvertCurrency". The provider responds with a request for the client to tell it the amount of money to be converted. When this is done it asks for the source then destination currency codes. If the client is unable to provide an amount then the service terminates with an error result. If the client does not understand the request for a currency code, the service can offer as an alternative, a list of codes for the client to pick from. After the provider has collected the information it needs, it processes the input and returns the result to the client.

5.3.1 A conversation language

Messages are the core mechanism for exchanging information between two parties. In free form natural language dialogues the interpretation of the type, purpose and content of messages is a complex process. There are several ways the complexity of the interpretation process can be reduced to mitigate the cognitive load on participants [29, 52, 143].

- Explicitly state the intent of a message.
- Explicitly state the purpose or type of a message.
- Define the set of allowable responses for each message.
- Change from free form dialogue (where anyone can say anything) to directed dialogue where the type of messages that can be sent in a given context is controlled.

All of these techniques are employed in guided interaction.

Messages

The information contained in a guided interaction message is described in the message schema shown in figure 5.2 as an ORM diagram [83].

There are two types of message, external and internal. External messages are exchanged between services and their clients. Internal messages are used within the

guide or dialogue manager. In addition to containing its *intent*, *performative* and application dependent *content* each message contains contextual information.

The context dependent information includes conversation or process id's for correlation supplied by each party in the dialogue. Messages may also contain a message id and a reference to a previous message if appropriate. The identities of the sender and receiver of the message are also part of the contextual information.

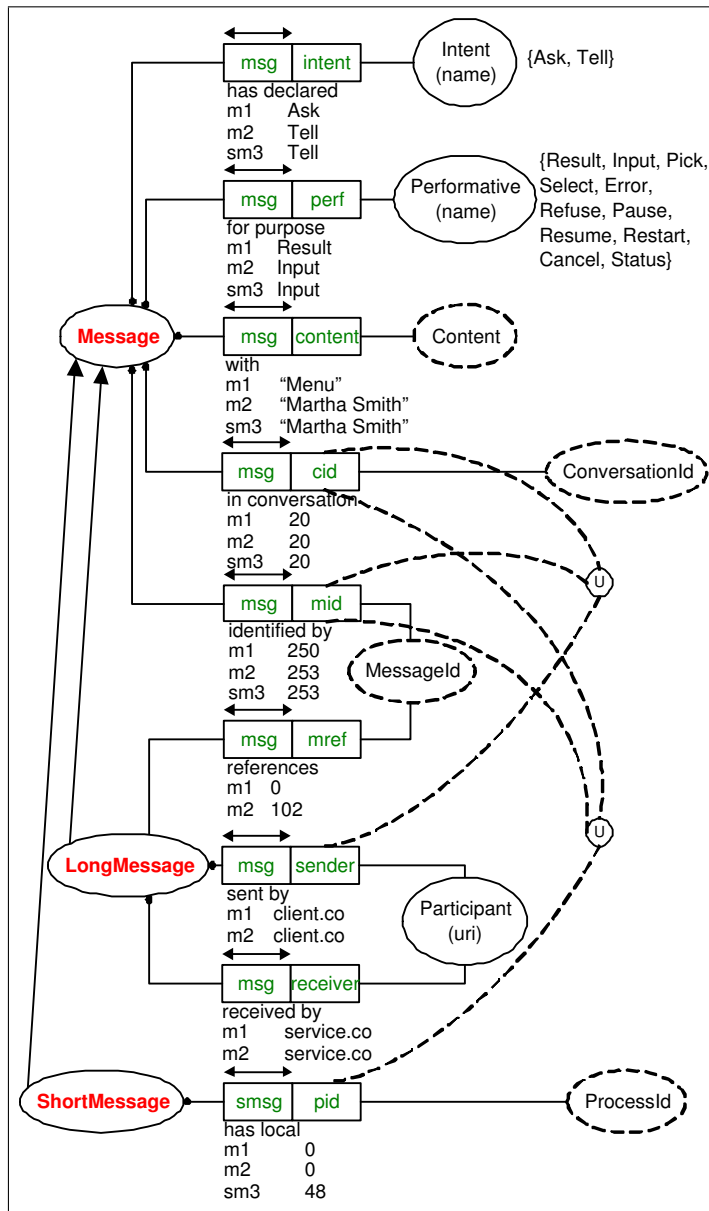


Figure 5.2: Message schema in ORM

The service provider generates a conversation id (pid) internally for each conversation because it cannot rely on external conversation ids being unique. For

example, two clients could use the same cid or the same client may reuse a cid. The provider generates a new pid to identify sub-dialogues of the main dialogue.

Each message is uniquely identified by a combination of the cid, mid and sender. An internal message is uniquely identified by a combination of a pid, cid and mid. These identity schemes are shown by the dashed lines connecting these elements to the uniqueness constraints (a circled U) in figure 5.2.

The actual structure of a message is flexible especially when implemented in XML, because in XML the elements can be accessed by name rather than position. An example of an XML definition and a sample message are shown in figures 5.3 and 5.4.

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.guided.org"
  xmlns="http://www.guided.org">
  <xsd:annotation>
  <xsd:documentation>
    Message schema
  </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleType name="ConversationId"><xsd:restriction base="xsd:anyURI"/></xsd:simpleType>
  <xsd:simpleType name="ProcessId"><xsd:restriction base="xsd:anyURI"/></xsd:simpleType>
  <xsd:simpleType name="MessageId"><xsd:restriction base="xsd:anyURI"/></xsd:simpleType>
  <xsd:simpleType name="ParticipantId"><xsd:restriction base="xsd:anyURI"/></xsd:simpleType>
  <xsd:simpleType name="Content"><xsd:restriction base="xsd:string"/></xsd:simpleType>
  <xsd:simpleType name="Intent"><xsd:restriction base="xsd:string">
    <xsd:enumeration value="Ask"/>
    <xsd:enumeration value="Tell"/></xsd:restriction></xsd:simpleType>
  <xsd:simpleType name="Performative"><xsd:restriction base="xsd:string">
    <xsd:enumeration value="Result"/>
    <xsd:enumeration value="Input"/>
    <xsd:enumeration value="Pick"/>
    <xsd:enumeration value="Select"/>
    <xsd:enumeration value="Help"/>
    <xsd:enumeration value="Error"/>
    <xsd:enumeration value="Refuse"/>
    <xsd:enumeration value="Status"/>
    <xsd:enumeration value="Pause"/>
    <xsd:enumeration value="Resume"/>
    <xsd:enumeration value="Restart"/>
    <xsd:enumeration value="Cancel"/></xsd:restriction></xsd:simpleType>

  <xsd:element name="Message">
  <xsd:complexType>
  <xsd:all>
    <xsd:element name="cid" type="ConversationId"/>
    <xsd:element name="mid" type="MessageId"/>
    <xsd:element name="mref" type="MessageId"/>
    <xsd:element name="sender" type="ParticipantId"/>
    <xsd:element name="receiver" type="ParticipantId"/>
    <xsd:element name="intent" type="Intent"/>
    <xsd:element name="perf" type="Performative"/>
    <xsd:element name="content" type="Content"/>
  </xsd:all>
  </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

Figure 5.3: XML Schema for Messages

```

<?xml version='1.0' ?>
<msg:Message xmlns:msg="http://www.guided.org"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.guided.org/Message.xsd">
<cid>234</cid> <mid>445</mid> <mref>232</mref>
<sender>client.co</sender> <receiver>service.co</receiver>
<intent>Ask</intent> <perf>Result</perf> <content>Menu</content>
</msg:Message>

```

Figure 5.4: Message example in XML

ORM is used in preference to XML because of its ability to visually present the information and constraints in a succinct format with a sample population. Figures 5.2 and 5.3 demonstrate how straightforward the translation from ORM to XML is, and an earlier example (section 3.4) showed how an ORM schema can also be translated to the ontology language OWL (shown in appendix A).

A small difference between the ORM schema and the XML schema, is the use of URIs⁹ to identify conversations, processes and messages in the XML schema. The ORM schema, presented in section 5.4, uses integer ids. This simplifies the implementation but real world applications would use URIs to identify these elements.

The remaining elements contained in the message, Intent, Performative and Content, are introduced in the next sections.

Intent

In section 5.2.1 it was shown that there were only two reasons for communicating: to ask questions and to tell answers. This is true for any software interface from command lines to windows. A client, be they human or software can only ask for information or actions, and tell information or the result of actions, and a provider (human or software) can only ask for information or actions and tell information or results. This was also noted in section 5.2.2 where it was found only three types of sentences are used in a technical context; interrogative, imperative and declarative. Interrogative and imperative sentences ask for information or actions and declarative sentences tell information or the result of actions.

This leads to the definition of the *Intent* of a message, Ask or Tell. It is not distinguished at this level whether a request is for information or actions. A conversation is the exchange of Ask and Tell messages with two constraints:

- C 1 For each Ask message there is only one corresponding Tell message in response.
- C 2 There is never a Tell message that is not in response to an Ask message

This mechanism does not “chat”.

⁹www.w3.org/Addressing/

An explicit statement of the Intent of a message allows both parties to understand their exact responsibilities in regard to the message. In addition, the use of only two intentions, mirrors the commonly used and well understood “Get” and “Set” operations of APIs and the “Get”, “Put” and “Post” operations in the REST architecture [58]. From an implementation perspective, both the client and provider require a very simple interface with only two operations, one for receiving Ask messages and the other for receiving Tell messages.

Performative

The purpose or type of the message is described by a *Performative*. The performative acts like a prompt, it indicates what kind of information is being sought or what kind of response is required. We propose a small set of performatives based on ACLs (section 5.2.3), agent communication protocols (section 5.2.4), and human computer dialogues (section 5.2.6). These performatives reflect core information gathering abstractions and dialogue management or control mechanisms.

- The **functionality** of a service is requested and delivered by the performatives *Result*, *Input*, *Pick*, *Select* and *Help*.
- The service **management** performative *Status*, provides information about the state of the dialogue at runtime.
- The **dialogue control** performatives are *Pause*, *Resume*, *Restart* and *Cancel*.
- The performatives *Error* and *Refuse* provide alternative responses.

Another way of looking at the list of performatives is to separate them into sentence types:

- Interrogative: *Input*, *Pick*, *Select*, *Help*, *Status*
- Imperative: *Result*, *Pause*, *Resume*, *Restart*, *Cancel*
- Declarative: *Error*, *Refuse*

The performative **Result** starts a conversation. It is sent by a client to the service to request the capability named in the content of the message. The intention behind the use of the word “Result” rather than “Do” or “Perform”, is that a dialogue is initiated by a client requesting the service to perform its advertised capability and to tell the result. So, a message containing *Ask*, *Result*, “*ConvertCurrency*” should be read as “Ask (for the) Result (of performing the) *ConvertCurrency* (capability)”. Although this may seem a little awkward it is necessary to ensure every performative has exactly the same name for the request and its response to reduce ambiguity.

The performative **Input** requests input data for a single item (parameter) similar in function to a text box or form item. As it is unrealistic to assume that two heterogeneous services will use exactly the same terminology and data structures, the focus is on describing what kind of content is required and the datatype the service expects rather than the value of content (as done by VXML grammars).

Clients have to match the provider’s request for input with the data they hold. If the client does not understand the terms used in the request they can ask for

Help from the service. This means that extended item definitions are necessary to allow the service to offer alternative information. The extended definition should include, in addition to the name of the parameter, a set of alternative names that are *equivalent in this context*. This information is found in the outsourced type descriptions described in chapter 3. Alternative names for the datatype of the parameter can also be provided. A sophisticated application may offer pointers to other services that can convert or reformat data into a form it can use, or it may make use of these services directly.

The capabilities associated with the datatypes may also be included in the outsourced type description. Examples of the kind of alternative information that may be found in outsourced type descriptions or from other sources and used for the currency conversion service are given in appendix B.

Pick asks the client to select from a list of acceptable values such as (AUD, GBP, USD, ERD, NZD) for currency types. It is similar to a drop-down list box in Windows.

Select offers a menu of choices representing the capabilities the service can provide. This mechanism can be used to offer more finely grained capabilities than those offered in the service advertisement e.g. from “ConvertCurrency” to “ConvertUSDtoGBP” or “ConvertUSDtoAny”. Select could also be used with a *generic* interface to a service provider. For example, if a client sends *Ask Result* “Menu” to determine which capabilities a service can provide the provider could respond with an *Ask Select* “...” message containing a list of its capabilities.

Informally, the difference between Tell **Error** and **Refuse** is that in the case of an error, the service (or client) tried but *cannot* generate a correct response. Refuse means the service (or client) *will not* provide the required response such as in the case of privacy or security violations. The distinction may or may not be relevant depending on the context of the service provider.

The dialogue management performatives **Pause**, **Resume**, **Restart** and **Cancel** have the effect indicated by their names. Both participants (client and provider) can use these performatives. If, for example the provider sends an *Ask Pause* message to the client, the client does whatever is necessary and sends a *Tell Pause* (or *Error* or *Refuse*) message in reply. When the provider is ready to resume, it sends an *Ask Resume* message to the client and the client responds with *Tell Resume* (or *Error* or *Refuse*).

The **Status** performative interrogates the state of the dialogue rather than the back end processing of the capability. Although this is not a complete view of the service, it does enable reporting on whether the provider is waiting for input from the client, or processing the last response, or has passed the client input to the back end processor.

This set of performatives is context and content independent which allows them to be used for the collection of input data for services in any domain.

For each *Ask performative* message there are only three valid responses:

C 3 Tell (the same) *performative*, Tell *Error* or Tell *Refuse*.

The advantage of constraining the allowable responses is that dialogue control is simplified and the dialogue is predictable and manageable, two examples are shown in figure 5.5.

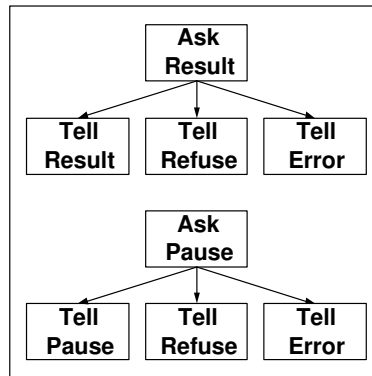


Figure 5.5: Two performatives and the allowed responses

In addition to the restrictions on performatives and their allowed responses, three constraints are imposed on the type of messages that can be sent by services or clients.

- C 4 Services as providers cannot Ask for a Result from a client, i.e. the service cannot ask its client to perform a capability while it is engaged in a conversation regarding the performance of a capability for that client. It could however, in the context of a new conversation request a capability from another service, which may or may not be its current client.
- C 5 Services and service clients cannot Ask for an Error or Refuse. These performatives are reserved for Telling the unsuccessful results of Ask requests.
- C 6 Clients cannot Ask for Input, Pick or Select from the service provider. If the client requires more information it can Ask for Help. In the implementation (section 5.4), help is provided based on the current item for which a value is being requested from the client.

These six constraints combined with the specified values for Intent and Performatives give the complete semantics of the dialogue language.

Content

In so far as possible the dialogue language and interaction mechanism are independent of the actual content of messages. Thus guided interaction is a generic dialogue model that is not tied to specific types of services or domains.

For the purpose of the implementation the content is defined as character strings. In a working application the content could be described in an XML schema.

A simple structure comprising a parameter name and datatype can be used for the input and pick performatives. Several of the other performatives lend themselves

to further structuring. For example a help request could contain the parameter name and datatype and the reason help is being sought e.g. not understood or not recognised.

5.3.2 A plan language

The previous section described *what* services can say to one another, but this is not enough for a complete language specification, also needed is a description of *how* the language is used in practice.

There are several activities that a dialogue management system should be able to perform particularly in the context of ad hoc heterogeneous service interaction.

- Provide help and disambiguation to mitigate data description mismatches.
- Provide alternative input sets, for services that can operate with different types of inputs, and to accommodate clients that cannot provide certain types of information.
- Use sub-dialogues to collect commonly recurring sets of input parameters such as credit card details.
- Provide context sensitive reports when fatal errors are encountered.
- Evaluate client input to determine the dialogue flow.
- Allow multiple concurrent dialogues about the same or different capabilities.

The rest of this section describes the operation of the guide or dialogue manager and the data structures used to hold information.

Instructions

Dialogues are directed the service provider using an internal plan to guide the collection of parameter information for a capability. A plan is a set of *instructions* detailing which inputs are required for a capability or operation. A client does not need access to the plan.

When the service provider receives and accepts an *Ask Result* message a capability plan is instantiated by inserting the cid from the clients request message and an internal pid into each instruction in the plan. In this way plans are uniquely tied to specific conversations, and multiple instances of the same plan can be activated concurrently. Processing starts with the first instruction in a plan.

Instructions have several parts as shown in the schema (figure 5.6).

The purpose of an instruction is to describe which item a value should be collected for. When an instruction is selected for processing (i.e. becomes the current instruction) the item it *references* is instantiated and a request message to the client seeking an input value for the item is generated.

The type of request message depends on the type of the item, this is discussed further in section 5.3.2. After the request message is sent the dialogue manager waits until a response is received from the client.

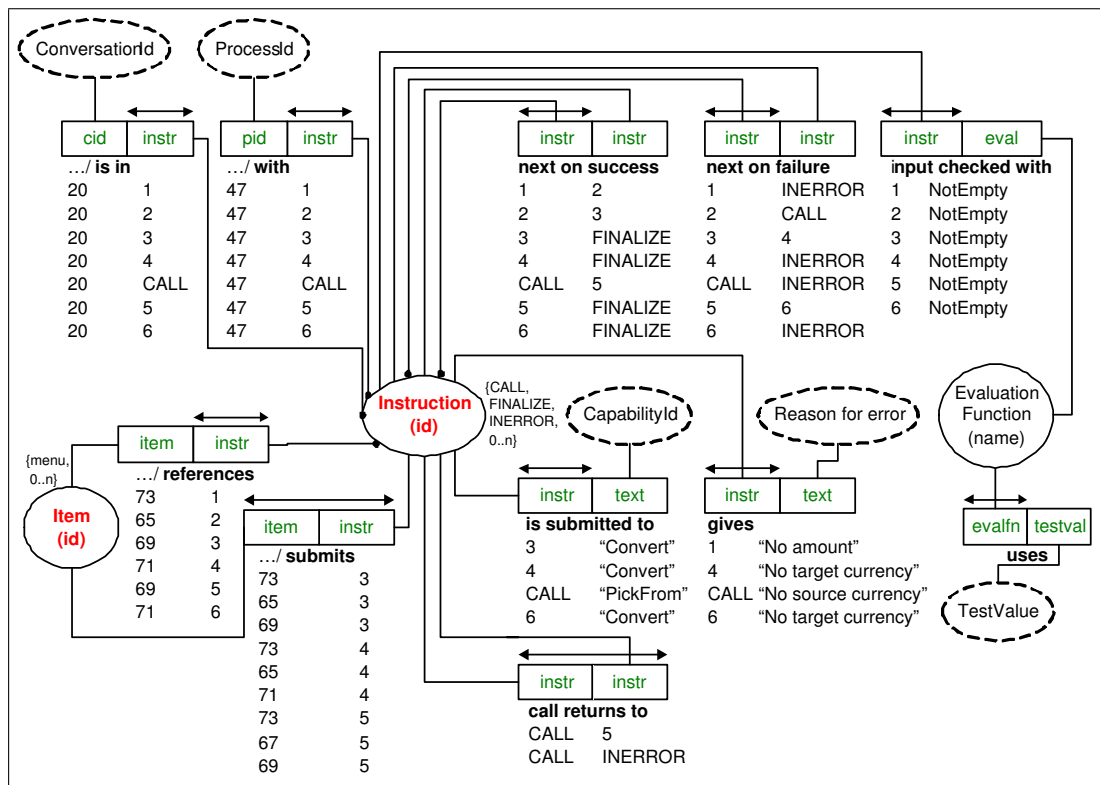


Figure 5.6: Instruction schema

An input value for the item received in a tell message is evaluated (*input checked with*) using the boolean *Evaluation Function* identified by name in the current instruction. If the evaluation returns true, the instruction identified by *next on success* is selected for processing and the input value is placed in the item and stored. If the evaluation fails the instruction identified by *next on failure* is selected for processing and the item is discarded.

The intuition behind only two choices of which instruction to select next (next on success and next on failure) is that input from the client is either valid or invalid. In the case of valid input the plan can proceed to the next step. In the case of invalid input there are two alternatives, either the item is critical to the provision of the capability and invalid input means the capability cannot be performed so a fatal error must be reported to the client. The other choice is to request input for an alternative item.

There are two ways alternative items can be used. The first is to ask the same question in a different way the second is to ask a different question. For example in the first case, when failing to elicit a value for a parameter when the name and datatype of the parameter were used to “prompt” the client, the plan can switch to asking the client to pick a value from a list of acceptable values. In the second case the plan could switch to a different input set, for example, a service that can

perform its capability with either a text file or a URL reference can switch to asking the client for a URL if they could not supply a text file.

In this way, the path of the dialogue is driven by what the service needs to know, which in turn depends on the information the client has been able to supply for previous items. The dialogue is driven by the remaining data requirements rather than an external conversation protocol or policy.

There are three special instruction ids, INERROR, FINALIZE, and CALL. An instruction with FINALIZE (as the *next* instruction) means the collection of inputs is complete and the values collected for the listed items should be submitted to the specified process or service. INERROR represents a fatal error that cannot be resolved by using alternative items. An INERROR instruction means the client is sent a message containing the reason for error detailed in the instruction and the dialogue terminates.

Figures 5.7 and 5.8 show that capability plans are binary trees with every branch of the tree terminating at a leaf node with a FINALIZE or an INERROR instruction id.

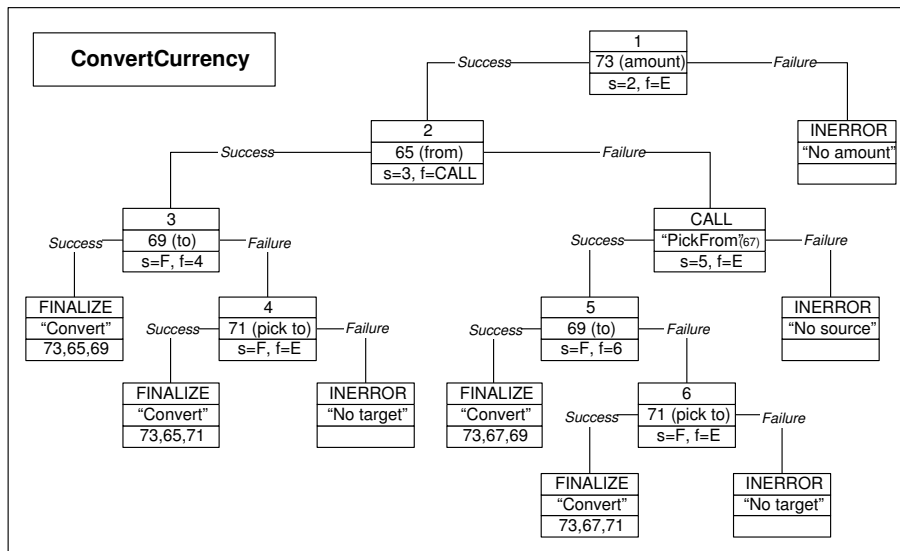


Figure 5.7: A tree representation of the “ConvertCurrency” capability plan

The other special instruction id, CALL, indicates that a subdialogue (representing another capability) is to be initiated. Subdialogues are modeled in the same way as dialogues, i.e. they terminate with FINALIZE or INERROR instructions. The advantage of not using an explicit return instruction is that all capability plans are described in the same way, i.e. not differentiating between main and sub-dialogues. This means a provider can call a capability such as “PickFrom” from within another plan (as a sub-dialogue) or expose the “PickFrom” capability directly to clients.

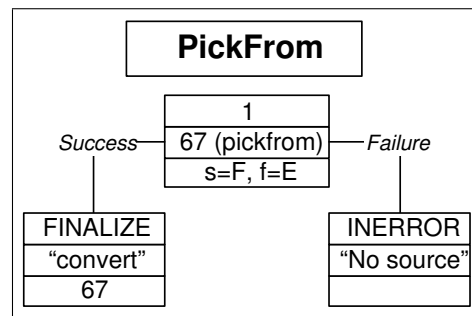


Figure 5.8: A tree representation of the “PickFrom” plan

An extension to the CALL mechanism could allow requests for a capability to be made to an external provider if the capability is not available internally. An implementation of this facility would require a mechanism to discover external services providing the required capability. The same evaluation procedure would apply to the results returned from an external provider and the dialogue would continue as specified in the CALL instruction. Use of the CALL mechanism to make external calls in this way shows how service *providers can become clients* of external providers while engaging in conversations with their own clients.

Calls to subdialogues are managed with a *CallStack*. The conversation id and process id of the process making a CALL are recorded in the CallStack along with the process id generated for the “called” process. When the dialogue manager encounters a FINALIZE or INERROR instruction id it checks the call stack to see if the instruction is terminating a previous call. If so, it removes the CALL record from the call stack and resumes the interrupted process using the cid, pid and the instruction id stored on the call stack.

The *ConvertCurrency* capability scenario introduced earlier is used to illustrate how instructions are used to generate requests, evaluate their responses and determine the next instruction to process. A tree view of the convert currency plan was shown in figure 5.7. An internal (CPN ML¹⁰) representation of this plan is shown in figure 5.9.

The cid from the clients request message and an internal pid are inserted into each instruction when the plan is loaded. In this way instantiated plans are uniquely tied to specific conversations, and multiple instances of the same plan can be activated concurrently.

The first line shows the instruction before being loaded

```
(1,0,0,73,((2,"",[]),(INERROR,"No amount",[]),(NotEmpty,"")))
```

. It is loaded with the conversation and process id’s resulting in

```
(1,20,47,73,((2,"",[]),(INERROR,"No amount",[]),(NotEmpty,"")))
```

¹⁰wiki.daimi.au.dk/cpntools/cpntools.wiki

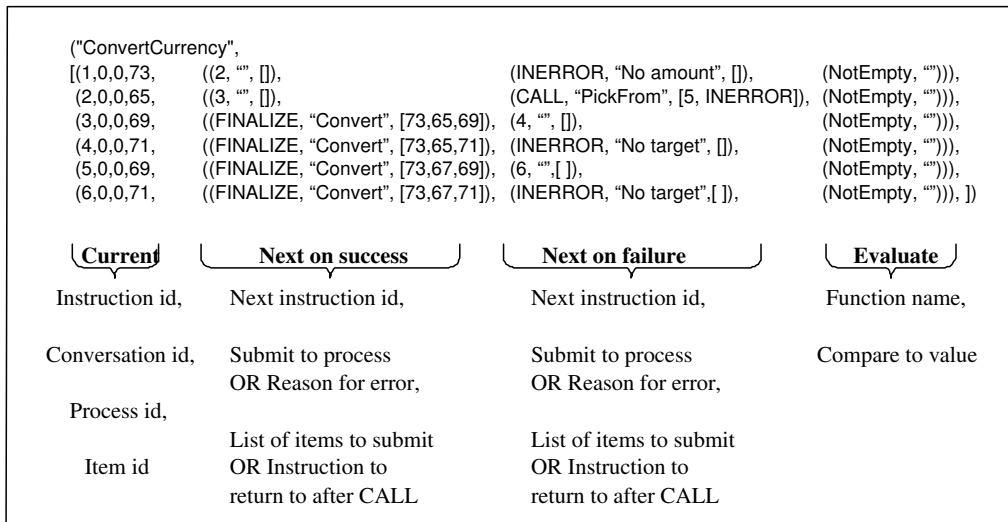


Figure 5.9: The convert currency capability plan

This is instruction number 1 in conversation 20, process 47. The item to be collected from the user is identified as item 73 (the amount to be converted).

The evaluation function named "NotEmpty" should be applied to user input and if the result is true then the statement

```
(2, "", [])
```

should be activated. This statement contains the id of the *next* instruction to activate after this one (instruction 2 in this case).

If the input evaluation fails (because the client did not or could not supply a useful value) the second statement is activated.

```
(INERROR, "No amount", [])
```

This signals a fatal error (INERROR) with the reason "No amount" because this capability cannot proceed without an amount to convert.

A path through the tree is illustrated by describing how to get to the FINALIZE node at the bottom of the tree where the items 73, 67 and 71 are submitted to the "Convert" service.

The guide initially gets an amount to convert from the client (item 73) and instruction 2 is selected for processing. Instruction 2 asks for a source currency code from the client (item 65 - from). The client cannot supply a value so the guide calls a sub-dialogue to gather item (67 - pick from).

A call is used here to describe how calls to sub-dialogues are made and return back to the calling dialogue. The sub-dialogue (shown in figure 5.8) uses item 67 to offer the client a list of currency codes to pick a value for the source currency. When

the client picks a value it is evaluated and the call terminates with a FINALIZE if the value is valid or INERROR if not.

The subdialogue returns successfully to instruction 5 (specified in the call). Instruction 5 requests the destination currency code (item 69 - to). As before if the client cannot tell a value the guide uses the alternative parameter (item 71 - pick to), which offers another pick list of currency codes to the client. A valid destination currency value returned from the client finishes the capability with the items 73,67 and 71 submitted for processing.

To summarise, the items necessary for the performance of a capability are gathered according to the order of instructions specified in the capability plan. The plan has a binary tree structure with each leaf of the tree either a FINALIZE or INERROR instruction. This structure allows the definition of alternative input parameters such as “Input” a value or “Pick” a value from a list. Alternative input sets can be defined such as “Input a flight number” or “Input an airline and a destination”. A plan can “call” another plan as a subdialogue.

The plan language allows developers to incorporate context sensitive help messages at each point in the plan where fatal errors can occur. This provides clients with information for focussed problem solving. Error information is also useful for reviewing operating performance and compliance checking.

Plans are structured as binary trees with each branch of the tree terminating at a leaf node with either a call to a back end process or a fatal error. Main dialogues and sub-dialogues are modeled in the same way. This allows sub-dialogues to be exposed to users as stand alone capabilities. It also allows capabilities to be requested from external sources.

The following summary shows how the plan language delivers the requirements outlined at the beginning of this section.

- The provision of help and disambiguation is enabled by the Help performative using alternative terms from the item descriptions.
- Alternative input sets can be described using instructions to gather alternative information when necessary.
- The modeling of all dialogues in the same manner means any capability plan can be run as a main or sub-dialogue.
- Context sensitive help can be included in the instruction to give specific information at the point of failure.
- All client input is evaluated to determine the dialogue flow.
- Multiple concurrent dialogues about the same or different capabilities are enabled by using a conversation id and a process id to identify each dialogue and sub-dialogue.

Obligations and expectations

Obligations and expectations are the means by which the guide maintains the context and state of multiple asynchronous conversations. The receipt of an Ask message generates an *obligation* to reply [4] and an *expectation* is created when Ask

messages are sent. Obligations are discharged when reply messages are sent and expectations are discharged when they are matched with replies.

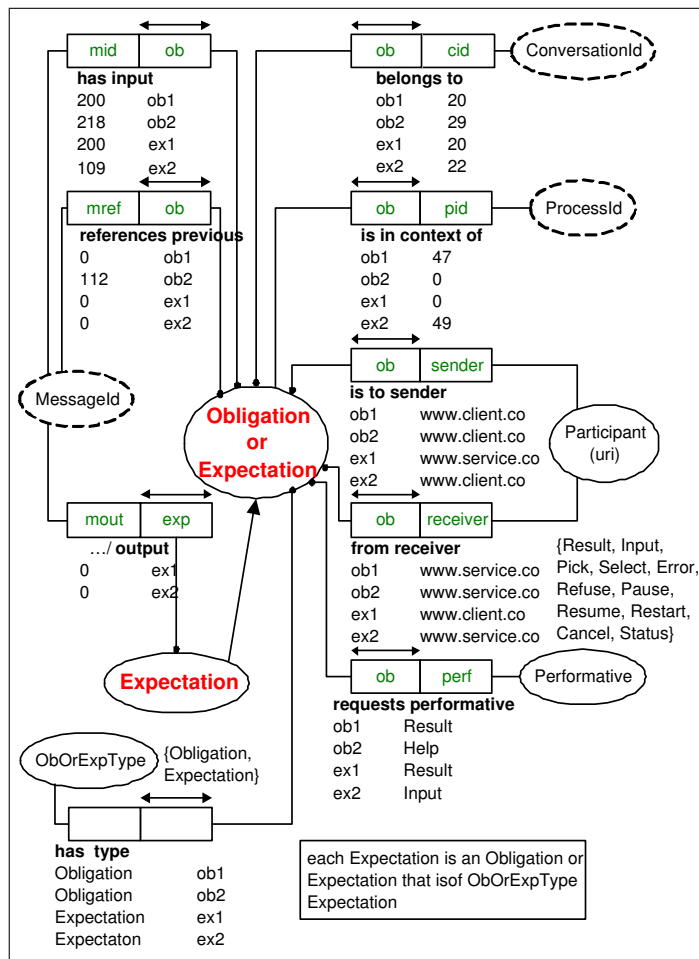


Figure 5.10: Obligation and expectation schema

An obligation (shown in figure 5.10), records the conversation id (cid), the id of this message (mid) and the id of a previous message (mref), if appropriate, from the received message. Finally, the identities (URLs) of the sender and receiver are recorded in an obligation along with the performative being requested. The performative is recorded to ensure the Tell message which eventually corresponds to this Ask message is a correct response to the request. Expectations have a similar structure to obligations, with the addition of the message identifier for the message being sent. Expectations keep a record of what information has been requested from the other party.

Items

The input request sent to a client is similar in function to a prompt. The type of request (Input, Pick or Select) and the information it should contain are determined

by examining the item identified in the current instruction. It is the type of the item that determines what type of request message is sent to the client.

Items are containers for parameters. The separation of items and their parameters allows parameter descriptions to be reused in different items. The item can provide additional information about a parameter when it is relevant in the context of a particular capability. Much of the detail in parameter and item descriptions results from using the outsourced types described in chapter 3. A conceptual schema for items is shown in figure 5.11.

An item, identified by an id, is instantiated in the context of a specific conversation. The *has been asked* role ensures clients are not asked twice for the same item. The item has two counters which are used to iterate through the lists of alternative names and datatypes specified in the parameter, when a client requests help.

The parameter (identified by name) belonging to an item contains these lists of alternative names and datatypes. The alternative names and types are used to provide names that are *equivalent in this context* to the client when they cannot match the term initially used by the provider. The parameter value is placed in the value slot when it is received from the client.

The ability to provide alternative names, grounded in external sources, is a key feature of this interaction mechanism. It is the means by which ad hoc disambiguation can occur. Instead of relying on shared definitions, each service is made responsible for providing equivalent alternatives for the terms it exposes to clients.

An item may contain one of two lists which provide context specific information. The first, is a pick list of alternative values that are relevant or appropriate for this item in this context. The second, is a list of menu options representing capabilities the service can offer. An item can not contain both lists as shown by the exclusion constraint between the *menu options* and *value options* roles in figure 5.11.

There are three performatives for input request messages Select, Pick and Input.

- An *Ask Select* message asks the client to select one of the offered capabilities. An Ask Select message is generated from an item that contains a menu options list.
- An *Ask Pick* message asks the client to pick a value from the list of values provided. An Ask Pick message is generated from an item that contains a value options list.
- An *Ask Input* message requests a data value from the client. An Ask Input message is generated from an item description that does not contain a value options list or a menu options list.

When a value is received from the client in response to an Ask Input or Ask Pick message, if the value is validated it is inserted into the parameter's value slot and the item is stored for later use. If the value is invalid the item is discarded.

The client also has a collection of parameters (or items) that hold the values it has obtained from other sources. There are several sources that could have supplied these values. The client may have obtained data directly from a human user. It may have had values provided during instantiation stored in parameters. It may have

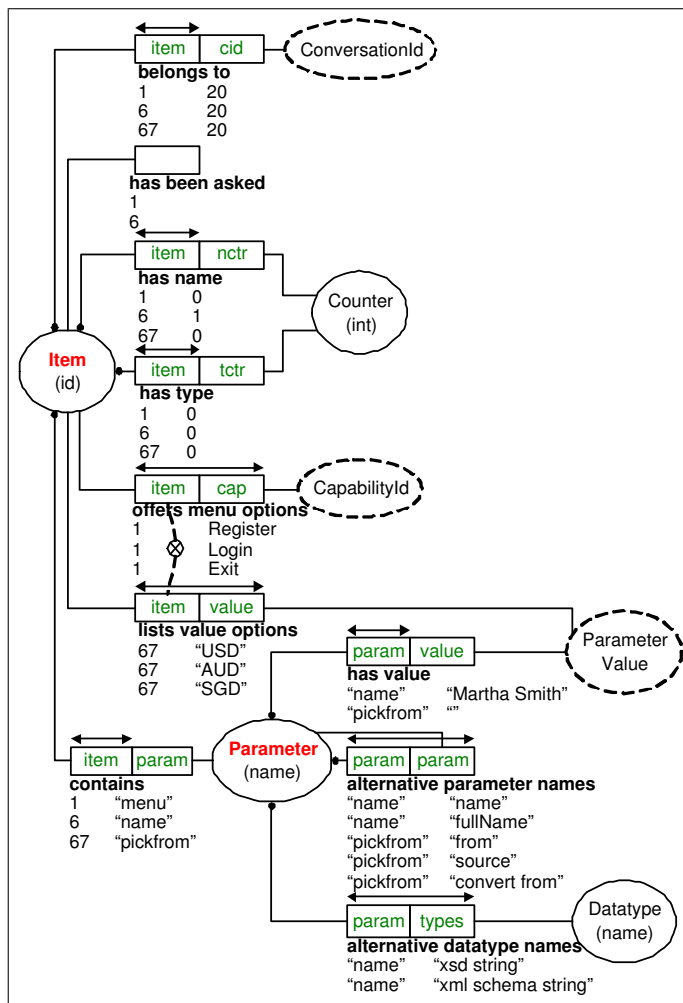


Figure 5.11: Item and parameter schema

received parameter values as the output of previously requested capabilities. Finally, the client could have received parameter values in the delivery of a capability to a client of its own, in the same way that our provider is receiving values from this client.

A client receiving an *Ask Input* message from a service will check within its list of parameters for a matching name and datatype. If the client can match the request with one of its parameter names, it will send a *Tell Input* message containing the parameter's value as the content. If it cannot match the request to a parameter name directly, it will have to look into the lists of alternative names contained in its parameters.

If the client cannot make a match between the provider's request and any of the names in its collection of parameters, it can send an *Ask Help* message to the provider, to get an alternative name (or datatype). This process can be repeated

until a match is found or the list of alternative names is exhausted. The failure to make a match means the client must return a *Tell Error* message.

The client also has to match the parameter name when it receives an *Ask Pick* message containing the name of the parameter required and a list of possible values. If the client matches the parameter name, it will then try to match the value it holds for that parameter with the list of values contained in the message. If a match is made the value is returned in a *Tell Pick* message, otherwise a *Tell Error* message is sent.

An *Ask Select* message contains a list of capability ids representing the capabilities that the service can provide. In this case, it is assumed the client has a list of *goals* that it needs to satisfy or a list of capabilities it requires. The client tries to make a match between the offered capabilities and its list of goals. There may be priorities associated with the goals which determine which one will be selected. The selection is returned in a *Tell Select* message or a *Tell Error* message is sent.

The representation of client goals and how these goals can be matched to the list of capability ids offered by the service is not addressed in this work.

Dialogue control messages

The Pause, Resume, Restart and Cancel performatives provide dialogue control for both clients and providers.

In the implementation semaphores were used to lock the output transitions when implementing the Pause and Resume performatives. An *Ask Pause* message sets the lock on all outgoing messages after the *Tell Pause* message which confirms the action. The receipt of an *Ask Resume* message removes the lock with all queued messages being sent after the *Tell Resume* confirmation message.

Implementing the Restart and Cancel performatives also required a lock on output messages, so a clean up process can be performed. Cleaning up after receiving an *Ask Restart* message from the client involves several operations. These include removing all obligations and expectations for this conversation except the first obligation. Removing all items that have been collected, and removing all the messages in the process of being sent to the client. The first instruction in the instruction set is reloaded. These operations are also necessary in response to an *Ask Cancel* message with the instruction set and all obligations for the conversation being completely removed.

5.4 Realisation: Tracing the delivery of a capability

This section describes in detail how the guide operates as it gathers the input it needs to deliver the convert currency capability. Section 5.3.2 detailed the data structures used by the guide and this section describes how the data is created, used and stored.