Guided interaction: A mechanism to enable ad hoc service interaction

Phillipa Oaks and Arthur HM ter Hofstede
BPM Program, School of Information Systems
Faculty of Information Technology
Queensland University of Technology
GPO Box 2434, Brisbane, QLD 4001, Australia

Abstract

Ad hoc interaction between web services and their clients is a worthwhile but seemingly distant goal. This paper presents *guided interaction* as an alternative to the current mechanisms for web service interaction which will allow heterogeneous web services and their clients to interact in a relatively simple and direct manner without pre-programmed calls to WSDL interfaces or human intervention at runtime.

Guided interaction is based on the exchange of messages that explicitly declare their intent and purpose. A service provider uses an internal plan to collect the input data it requires from the client in order to deliver its capability. It is the nature and sequence of the data requirements specified in the plan and the ability of the client to provide the data that determine the path of a dialogue rather than a pre-defined conversation protocol.

Clients do not have to know in advance a service's operation signatures or the order of operations. They can request further information about specific items as they are guided through the data input process. Dynamic disambiguation of terminology is an intrinsic feature of guided interaction.

1 Introduction

Automated ad hoc interaction between web-based applications is a desirable goal. Applications that can automatically locate and interact with software services without a priori knowledge of their interfaces will be able to achieve many tasks that are beyond human resources at present. These tasks can be as diverse as real-time monitoring and resource reallocation; repetitive polling for events that require a tactical response; and gathering of location dependent information for mobile devices.

The vision of real-time access to vast sources of information was described by Gelernter in 1991 [17]. Now, having achieved that vision with the Internet, we are looking at how this

evolving collection of information and resources can be accessed and used effectively. This is on the verge of becoming a reality with the development and proliferation of software services designed to be accessed by programmatic means (i.e. from within a program and without direct human intervention)[1]. The increasingly large number of software services means that their use needs to be automated to the largest possible extent in order to fully profit from the possibilities that they open.

Ad hoc is defined in Wordnet[2] as "unplanned" or "without apparent forethought or prompting or planning". In an ad hoc interaction environment, a software client could find, using for example a discovery mechanism, a software service that provides the capability it (the client) requires at that time or place. Depending on the mechanism used to find services, the client may have little or no knowledge about the inputs the service requires, the dependencies between the data inputs, the order of invocation of its operations, or the type and formatting information associated with these inputs.

Ad hoc interaction is very different from the present situation where software applications interact in a planned manner via interfaces. A software interface provides a static view of the operation signatures provided by a software service. Operation signatures detail the names of the operations and their input and output data types. If the client uses the correct name, provides the correct type of information for the parameters in the correct order and ensures any preconditions are satisfied, then the provider will do the operation in the promised manner. This means that software client programs, including web services, are pre-programmed to make "calls" to operations on the public interfaces of service providers.

Interfaces providing the same functionality can differ from one another in several ways. They can use different names for the same operations, they can use different names and data types for parameters, and they can require the parameters in different orders. This diversity is inherent to software and stems from developer idiosyncracies and enterprise culture and conventions. The result of this is that a client programmed to use the interface of one provider cannot switch at runtime to use the interface of another provider even if those interfaces provide the same functionality.

The Internet and the world wide web have now made many different types of information accessible on demand. The numbers of providers and the types of information becoming available mean that the current interaction mechanisms based on prior knowledge about software interfaces will not scale up to the potential benefits of ubiquitous web accessibility. This is especially true with the increased use of mobile computing devices which can move in and out of spaces that can house any number of context dependent and independent services .

There are three main approaches to address or avoid the problems associated with ad hoc service interaction:

1. Use standard interfaces which require all services providing the same functionality to use the same interface.

[1]When accessible over Web-based standards, such software services are usually known as "Web services".

[2]wordnet.princeton.edu/

2. Use protocols of interaction to prescribe the order of message exchanges.
3. Use techniques to obtain information about the operation signatures of software services at runtime such as dynamic CORBA and Java reflection.

In the standard interfaces[3] solution all providers of a particular function use the same interface. The solution relies on a common agreement between heterogeneous service providers on the best interface for a particular operation or set of operations in a particular domain.

The problem is that that standard interfaces limit ad hoc interaction in two ways. Firstly, it is the responsibility of the client application's programmer to know or find out how to call the operations supplied by the interface[4] in order to pre-program the appropriate invocations. Secondly, it locks all providers into the same signature. If the same task can be performed with the same inputs but formatted in a different manner or with different sets of inputs the provider must supply (non-standard) interface operations for each of these variants.

In reality, the number of service providers, the number of possible operations and the different contexts in which those operations will be performed means the standard interface solution will not scale to solve the problems of ad hoc interaction in the heterogeneous web services environment.

The second proposal is for the use of interaction protocols. Interaction protocols are message exchange plans that define the type of messages that can be sent, by whom, and in which order. The argument for interaction protocols is that if both sides are aware of the correct type and sequence of messages they can participate correctly in the interaction.

It is true that some conversations or dialogues follow repeatable patterns and for some problems in clearly defined contexts where the data requirements are well understood by all parties interaction protocols can be developed. A good example of this case is the English auction protocol.

Much of the work that has been done in the area of conversational interaction for software agents and lately web services is directed at specifying the order of messages in a conversation. These interaction protocols are defined as either state charts [21] with messages representing the transitions between states [20], as AUML interaction diagrams [30], or as Coloured Petri Nets (CPN) [10]. For web services, BPEL[5] and the Web Services Choreography Description Language (WS-CDL)[6] are XML based specifications for describing the order of message exchanges.

However, ad hoc service interactions will take place in a context where the nature and order of the information service providers require may not be known by clients. Unlike auctions, each provider of the same capability may require different types of data or they may use different terminology to describe the same data.

[3] www.learnxmlws.com/book/chapters/chapter11.htm

[4] Many common software development environments help the developer in this task by reading the service interface and automatically generating the code necessary to interact with the service. However, these tools are only usable if the interface of the service is available when the client application is developed.

[5] `www-106.ibm.com/developerworks/webservices/library/ws-bpel`

[6] `www.w3.org/TR/ws-cdl-10`

Ontologies for protocol description have also been proposed such as the one in [35] and another found at `http://www.csl.sri.com/users/denker/sfw/wf/ip.owl`. Ontologies provide a way of "serializing" protocol descriptions into the Web Ontology Language[7] (OWL) for sharing across the web. However, just as these two example ontologies have different underlying conceptual models, so to, the protocols created using these ontologies will differ depending on the needs, requirements and preferences of the parties defining the protocols.

A problem with all of these approaches is how the interaction protocols are shared, understood, agreed upon, and enacted at runtime. The computational cost associated with the effort required to agree on which protocol to use [31] and to ensure runtime compliance [36] without human intervention is a serious problem that has not been addressed.

The lack of a shared understanding of the data requirements (as there is in the auction context) means ad hoc interaction between heterogeneous services cannot be fully defined in terms of a pre-set pattern of message exchanges. It is the nature and sequence of the data requirements of each service provider and the ability of each client to provide the data that should determine the direction of a dialogue.

The third possible approach to enable ad hoc interaction is the use of techniques to obtain information about the operation signatures of software services at runtime. These techniques are used by client programs to gather information such as the names of operations and the data types the operations expect as input and return as output.

One of the difficulties of doing this on a large scale is that the information that is available at runtime via reflection is syntactic rather than semantic. Client programs need to interpret this derived syntactic information and create semantically correct request objects or messages to send to the provider at runtime. This interpretation effort places a large computational burden on the client at runtime supported by the developer programming the client software with context dependent discovery, interpretation and message generation logic at development time.

The many diverse kinds of software services that will be developed in the coming years and the promise and potential of ubiquitous service access means that the current paradigm of pre-programmed one-on-one interactions with providers whose interfaces are known at development time will not provide 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". This mechanism must reduce the computational burden on client programs allowing them rapid and flexible access to any service that can fulfil their current needs.

This paper introduces a meta protocol called *guided interaction* that does not specify what data should be exchanged as in standard interfaces, or specify the order of messages as in interaction protocols, or require intensive computational effort on the part of client programs. Guided interaction defines a language and rules for conversational interaction between services [26].

Guided interaction uses a shared language for the exchange of information in messages.

[7] `www.w3.org/2004/OWL/`

These messages explicitly declare their intent and purpose. As in all computer interaction, the intent of a message is to either *ask* for information or for an operation to be performed, or *tell* an answer or the result of performing an operation. The purpose of a message is described with a performative that describes the kind of information being sought or sent. The intent, a limited set of performatives and six constraints provide the complete semantics of the shared interaction language.

A shared language for interaction is not sufficient on its own, it must also be possible to interpret, manage, and generate messages in the language [12], such that the result of the interaction satisfies goals or solves problems. Guided interaction also defines a way to create dialogue plans which allow a plan interpreter to generate and interpret messages in the language and manage the flow of each dialogue. These dialogue plans are an internal structure of a service provider, clients do not need and do not have access to the plan.

Guided interaction is designed to facilitate 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 guide or service may not need to use this mechanism for repeated interactions.

The purpose of guided interaction is to facilitate communication between two entities that have no prior knowledge of one another. It is assumed however that the client is aware of the function or information the service delivers which can be found in a published "capability" description [28]. This means 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. The functionality of the service is fixed and it is the responsibility of the client to find services that promise the required functionality, it is not the services' responsibility to adapt their functionality to serve the transient needs of individual clients. The (AI) planning concerns of how client goals are represented, how they are decomposed for discovery and matched to the capabilities delivered by service providers is out of the scope of this work.

Guided interaction means clients do not have to know in advance a service's operation signatures or the order of operations before asking the provider to perform a capability. However, the client should have access to appropriate data before engaging with the service. For example, before engaging a bank service a client will have information relevant to banking such as account numbers and transaction amounts. Clients can request further information from the provider about specific items as they are guided through the data input process. Dynamic disambiguation of terminology is an important feature of guided interaction. A means of facilitating shared understanding of the syntax and semantics of the terms used by the service provider are essential for loosely coupled ad hoc interaction.

The focus of guided interaction is not on specifying the order in which messages are sent, but on specifying the information that is (still) required before the service provider can execute the required service operation. It is the nature and sequence of the data requirements specified in the plan and the ability of the client to provide the data that determine the path of a dialogue rather than a pre-defined conversation protocol. The service provider can use alternative ways of asking for input data and it can use alternative data sets to provide a flexible interface for clients with different competencies and data

holdings.

The rest of the paper is structured as follows. The foundations section (section 2) outlines several technologies which have influenced the definition of guided interaction. Section 3 presents the details of guided interaction. Section (4) reviews several other proposals for web service interaction. Paper concludes with a discussion in section 5.

Please note, several of the footnotes contain URL's the string "http://" has been removed to save space and should be pre-pended to all URLs before attempting to access them in a browser.

2 Foundations

Various mechanisms, paradigms and technologies have contributed to guided interaction. Several questions guided the exploration of the notion of ad hoc interacting services, 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? In this section the technologies that have contributed to the answers to these questions are given a brief introduction and the ideas which have influenced the proposals that follow are outlined.

Computer interaction mechanisms: In the early days of personal computers human computer interaction involved using the command line interface [32]. 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. Parameters are accessed by position not by name so users have to know the command name and the correct sequence of parameters in advance.

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 the appropriate information 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.

Point and click interfaces have greatly improved access to software for most people, however they have not introduced any new mechanisms for gathering the input from the user. These interfaces are still based on three input mechanisms: input of a single data item, pick from a list of values and select a command from a menu representing an action to be performed.

Linguistics: In linguistics four types of sentence, declarative, interrogative, imperative and exclamatory[8] are recognized. Declarative and exclamatory sentences declare facts,

[8] `www.uottawa.ca/academic/arts/writcent/hypergrammar/sntpurps.html`

interrogative sentences ask questions and imperative sentences give commands. Usually only three (declarative, interrogative and imperative) are used in a technical context[9].

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 computer interaction mechanisms i.e. ask for input information, ask for an action by selecting from a menu of commands, and tell answers or results.

Intelligent Agents: Intelligent agents are proactive, reactive, autonomous goal seeking, communicative, and possibly mobile software entities [14, 7]. They use Agent Communication Languages (ACLs) to "talk" to one another. There are two primary agent communication languages FIPA ACL [13] and KQML [27, 24].

KQML performatives and FIPA ACL Communicative Acts (CAs) are derived from speech acts [34]. Performatives or CAs 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.

ACL *messages* contain three types of information.

1. Contextual information 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 ideas taken from ACL's are the ability for software entities to communicate using a "conversation-like" exchange rather than a more typical "operations on interfaces" type of interaction and the structured format for messages.

There are several performatives that are relevant for one to one service interaction. These are the 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 discussion[10] 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 in the discussion suggest there is a convergence between agents, services and semantic web services. The guided interaction mechanism presented in section 3 will further assist this convergence by providing a means of communication to bridge the gap between the agent and service interaction mechanisms in use at this time.

[9] `mit.imoat.net/handbook/s-types.htm`

[10] See `sharon.cselt.it/projects/jade/jade-develop-archive/0321.html`

Protocols: Performatives and the set of appropriate responses to them can be seen as the building blocks for protocols. Burmeister et.al. [8] use three basic "building block" performatives *inform*, *query* and *command* to build protocols of interaction.

The responses 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 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.

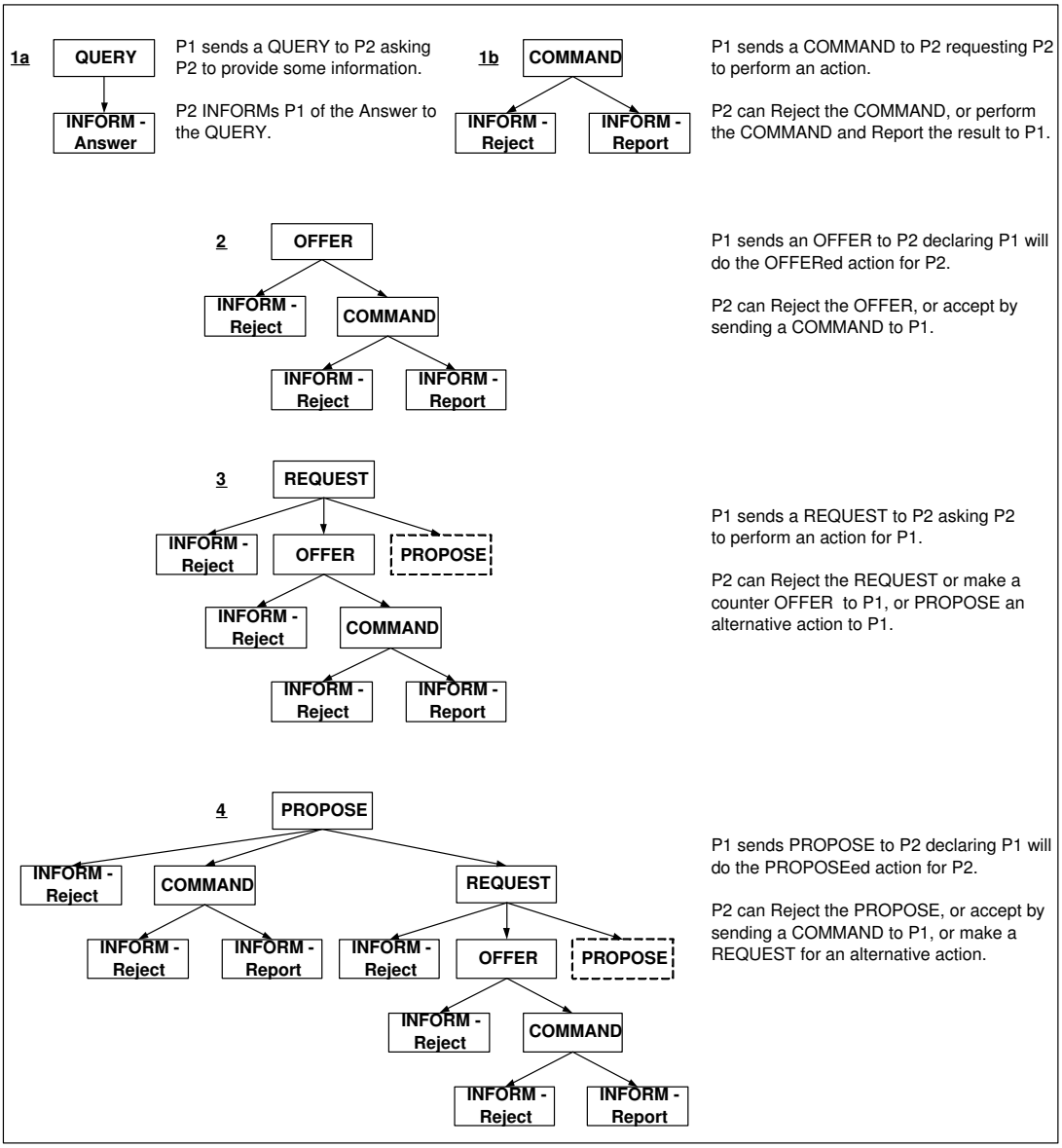

Figure 1: Protocol building blocks

Complex protocols can be built from these basic building blocks as shown in figure 1. The two basic protocols, 1a and 1b, are shown at the top of, with 2,3 and 4 being new protocols defined in terms of these two basic protocols.

Two of the ideas from Burmeister et.al. incorporated in the guided 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 computer interaction. The second idea is that a protocol or performative is defined in terms of the responses that can expected when it is used.

Wizards: Wizards are software programs that guide users through complex tasks, or tasks that may have many steps and require those steps be taken in a prescribed sequence [6]. Wizards are particulary useful for novice users who lack the necessary knowledge to perform a task.

The main idea drawn from wizards is a view of service clients as equivalent to novice human users. Wizards are a concrete demonstration of how complexity can be broken down into a sequence of simple interactive steps.

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 [16] in which the dialogue mechanism shields the client from back end applications.

Dialogue architectures are structured around the performance of three functions. The first function is interpreting (spoken) user input, the second is managing the conversation and the third function is generating output for the user. Allen et.al. [1] hypothesize 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 receive, interpret, and send messages it can participate in any dialogue regardless of the actual content of specific conversations.

Voice XML: VoiceXML[11] (VXML) is a specification for 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 a document model with each document describing the information to be gathered in the dialogue. VXML uses two primary abstractions for collecting information, forms and menus.

Forms hold 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

[11] `www.w3.org/TR/voicexml20/\#dmlAFIA`

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.

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 similar to those in section 2. The structured specification of dialogues in XML for processing by the VXML engine is a useful implementation technique.

3 The guide: a dialogue mechanism for services

Guided interaction is based on a conversational or directed dialogue model [22]. 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. 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.

Guided interaction has two parts. The first part, introduced in section 3.1 is a shared language for the exchange of information between services. The second part, introduced in section 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.

There are several ways guides could be used: a guide could be provided as an alternative means of accessing one or more capabilities represented by WSDL specifications or other APIs. Guides could be implemented directly by service providers or by independent third parties using publicly accessible interfaces.

A guide has been implemented as two communicating Coloured Petri Nets (CPN)[12] shown in appendix A. CPN uses the CPM-ML language for declarations and net inscriptions.

Two scenarios are outlined below which 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 (a PDA) has been directed to a service that converts an amount from one currency to another. The PDA/client is unsure of the exact terms

[12] `wiki.daimi.au.dk/cpntools/cpntools.wiki`

the provider uses to represent its capabilities so it asks the provider to provide 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. If the client is unable to provide an amount then the service terminates with an error result. Once the provider has the amount it asks for the source then destination currency codes. If the client does not understand the request for a currency code the service can alternatively ask the client to pick a currency code from a list. After the provider has collected the information it needs, it processes the input and returns the result to the client.

3.1 A language for service conversations

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 [8, 11].

- 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.

Each of these techniques is employed in guided interaction.

The information contained in a guided interaction message is described in the message schema shown in figure 2 as an ORM diagram [19]. ORM is used because it is a highly expressive conceptual modeling language that allows the visual presentation of information in a succinct format with a sample population.

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 ids and a references to a previous messages if appropriate. The identities of the sender and receiver of the message are also part of the contextual information.

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. A new pid is generated 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 2.

A suggested message format is described in the message schema shown in figures 3 and 4. 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.

Figures 2 and 3 demonstrate how a straightforward translation from ORM to XML can be made.

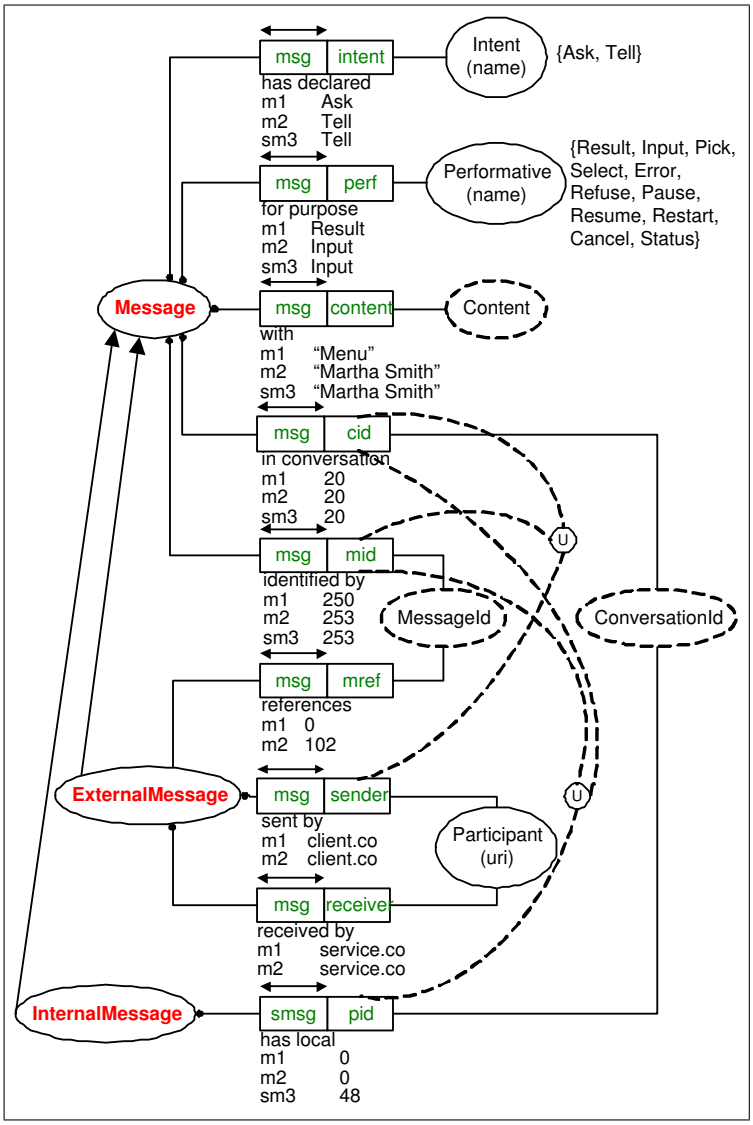

Figure 2: Message schema

```
<?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 3: XML Schema for Messages

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

3.1.1 Intent

In computer interaction there are only two reasons for communicating: to ask questions and to tell answers. This is true for any software interface from command lines to windows and APIs . 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 is also seen in linguistics where only three types of sentences are used in a technical context, interrogative, imperative and declarative. Interrogative and imperative sentences request either 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. At this level it is not distinguished 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 A Tell message can only be sent in response to an Ask message.

This mechanism does not "chat".

An explicit statement of the Intent of a message allows both parties to understand their exact responsibilities in regard to the message. 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. 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.

3.1.2 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

```
<?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 4: Message example in XML

of response is required. A small set of performatives is defined based on ACLs, agent communication protocols, and human computer dialogues (section 2). 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.

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", "Perform" or "Start" is, 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.

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 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 guide. This means every internal parameter definition should include a set of alternative parameter names and datatypes which are *equivalent in this context*. This will allow the provider to offer alternative information to the client. Ideally these names and their alternatives a references to web accessible sources such as public ontologies or dictionaries such as Wordnet [29]. A more sophisticated provider may offer the client pointers to other services to convert or reformat data into a useful form, or it may make use of these services directly.

The performative **Pick** asks the client to select from a list of acceptable values such as (AUD, GBP, USD, ERD, NZD) for currency codes. Its function is similar to a list box in Windows.

Select offers a menu of choices representing the capabilities of the provider. Select can be used to offer more finely grained capabilities than those described in a 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 an error is returned if 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 concerns. The distinction depends on the context of the participants.

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 an incomplete 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.

An important feature of this set of performatives is that they are context and content independent. This allows them to be used for the collection of input data for services in any domain.

There is one constraint that applies to all performatives except Error and Refuse. 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 the dialogue is predictable and manageable.

There are three constraints on the types of messages that can be sent by providers or clients.

C 4 Service providers cannot Ask for a Result from clients, i.e. a service cannot ask its client to perform a capability while it is performing a capability for that client. It could however, in the context of a new conversation request a capability from another service, which may be its current client.

C 5 Providers and 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.

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

3.1.3 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. Implementations could use XML for the content of messages, with the structure described with 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 recognized.

3.2 A language for conversation plans

A vocabulary for interaction was described in the previous section but a complete language specification also needs a description of *how* the language is used in practice.

There are several activities it is desirable a dialogue management system should 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 who 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 data structures necessary to hold information within a dialogue manager.

3.2.1 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). 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 3.2.3. After the request message is sent the dialogue manager waits until a response is received from the client.

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 place 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 instruc-

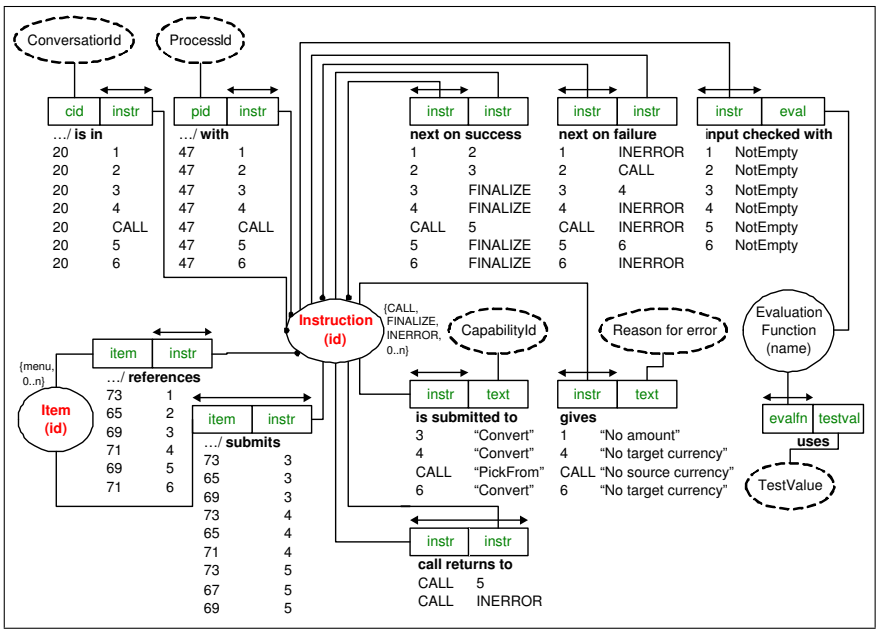

Figure 5: Instruction schema

tion 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 6 and 7 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.

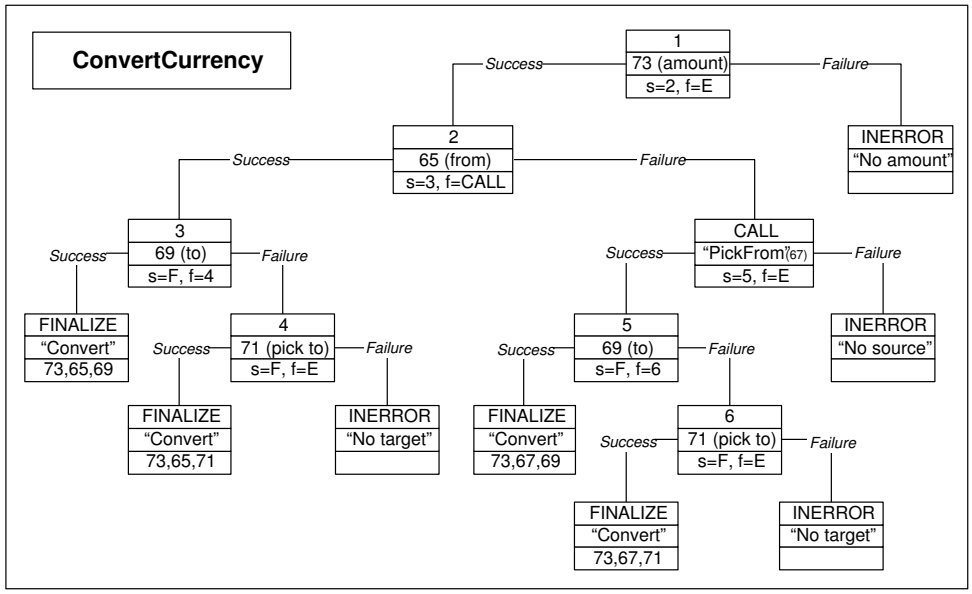

Figure 6: A tree representation of the "ConvertCurrency" capability plan

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.

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 is shown in figure 6.

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 7) 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.

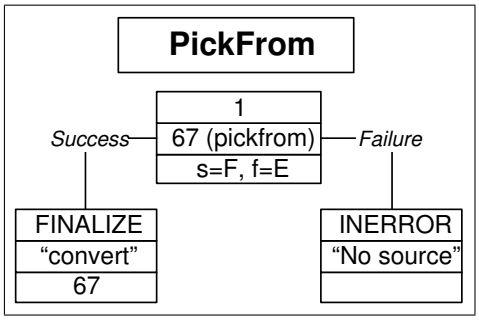

Figure 7: A tree representation of the "PickFrom" plan

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 summarize, 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.

3.2.2 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 [2] 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.

An obligation (shown in figure 8), 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.

3.2.3 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 used to hold 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. A conceptual schema for items is shown in figure 9.

An item, identified by an id, is instantiated in the context of a specific conversation. The item has two counters used when a client requests help to iterate through the lists of alternative names and datatypes specified in the parameter.

The parameter belonging to an item contains the lists of alternative or substitutable names and datatypes as discussed in section 3.1.2. The alternative names and types are

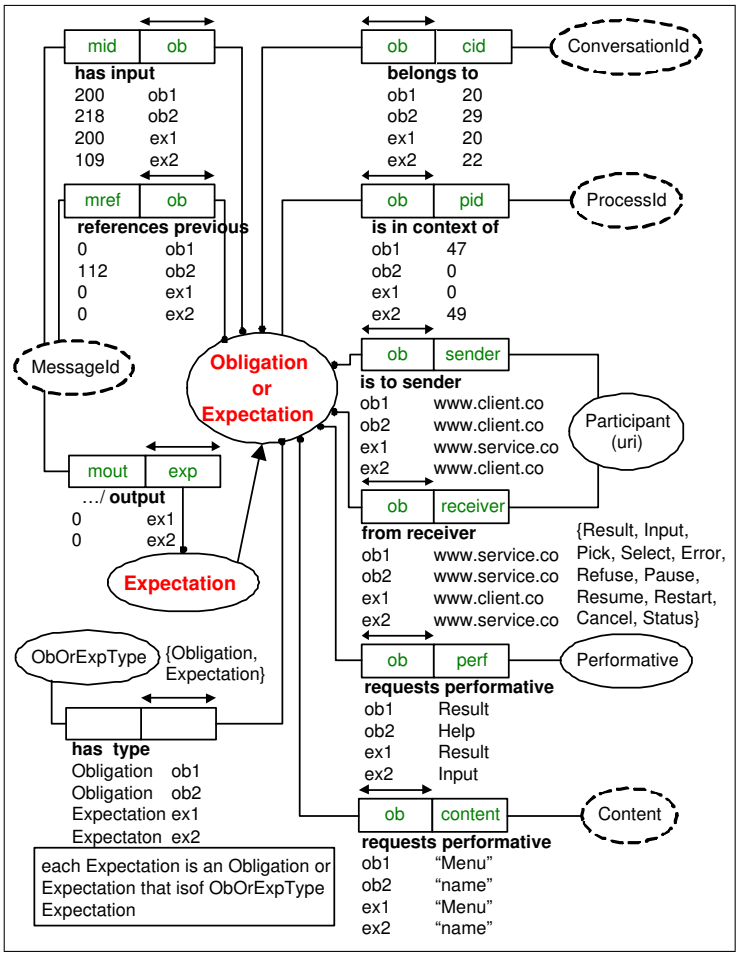

Figure 8: Obligation and expectation schema

those that are *equivalent in this context*. The parameter value is placed in the value slot when it is received from the client.

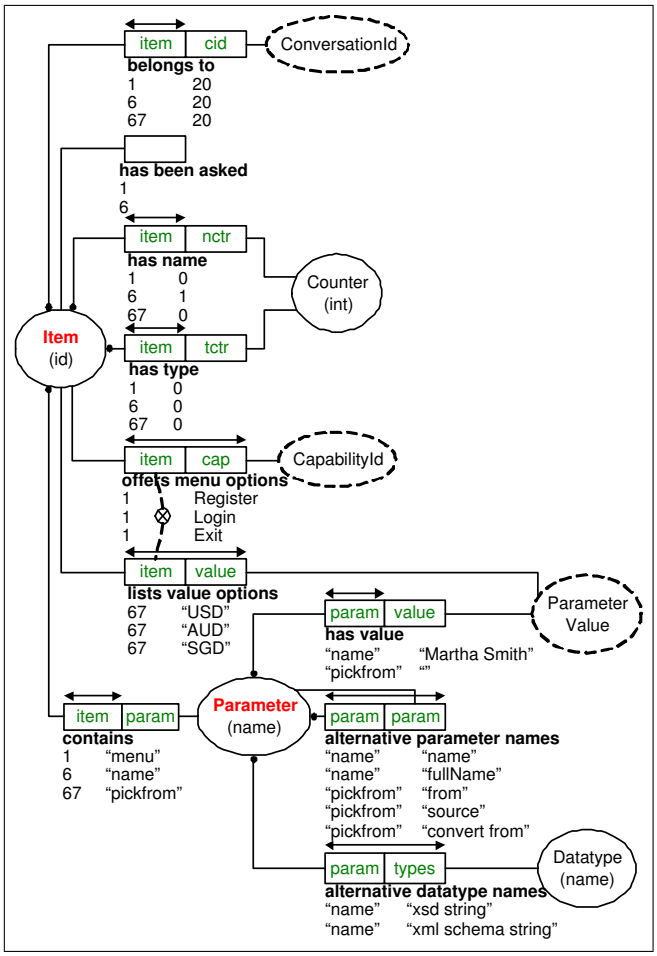

Figure 9: Item and parameter schema

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 9.

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 containing 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 containing 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.

A client also has a collection of parameters (or items) which hold the values it has obtained. There are several sources a client may have obtained data values from. The client may have obtained data directly from a human user, or the client may have had data values provided during instantiation. The client may have received parameter values as the output of previously requested capabilities. Finally, the client could have received parameter values from a client of its own, in the same way this 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 providers 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 the service can provide. In this case, it is assumed the client has a list of *goals* 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.

3.3 Future work

Guided interaction is designed to facilitate interaction between software entities that have not been explicitly pre-programmed to interact with one another. The dialogue mecha-

nism as implemented asks separately for each input parameter value. This is appropriate behaviour with clients who have no prior knowledge of the service's requirements and when item specific help may be necessary but it can be an inefficient mechanism for repeated interactions.

Future efforts could be directed at optimizing of the collection of inputs for clients who have prior knowledge of the services requirements for example, from the input signature definitions in the service's capability advertisement. These clients could be asked for input for several items in one message rather than individual requests as at present.

Another interesting avenue for further work would be to explore how a guided interaction could be used for software clients to "learn" how to interact with a service. Clients could learn enough during a guided interaction to be able to use the more traditional, and efficient, interface based means of communication for repeated interactions.

Capability plans are a list of zero or more input items to collect in a more or less sequential order. A more sophisticated processor could allow sets of items to be submitted for intermediate processing and modifications to the current plan depending on the results. If new or altered plans are adopted during processing the new plan would need to specify if previously collected items are reused or collected anew.

4 Related work

IBM's Conversational Support for Web Services (CPXML) [20] describes Conversation Policies (CP)[13] which are a state chart rendered in an XML document with elements that describe the state and transitions involved in a conversation. There are three possible states: normal, inchild or terminal. Message transitions detail which message is being sent or received. Transitions, to and from sub-dialogues (child states) are treated in a special manner. Although there is a good motivation for service conversations, including "peer-to-peer, proactive, dynamic, loosely coupled interaction" this is not clearly realized by the specifications.

The SELF-SERV platform is the implementation vehicle for the conceptual modeling of conversations described in [4]. This mechanism also uses a state chart based representation. A conversation manager, implemented on the SELF-SERV platform, uses the state chart representation of a conversation which extracted into a control table. The control table associates the conversation states and transitions with events, conditions and actions (ECA Rules). Transitions can be explicitly triggered by messages or implicitly by internal actions. The nature of the messages is not elaborated but it seems likely these are messages defined in WSDL. Although there is the possibility that states may not be exited due to a failure to satisfy the conditions or other errors this situation is not addressed. There is no facility for runtime disambiguation if clients do not understand the service providers terminology. Clients have no control over the management of the dialogue, such as requests to pause and resume etc.

The Web Services Conversation Language (WSCL) from Hewlett Packard [23, 15] models the conversation as the third party in an interaction. A conversation controller

[13] `www.research.ibm.com/convsupport/examples/ConversationPolicy2.0.xsd`

keeps track of a conversation and changes state based on the types of messages i.e. the type of the XML document the message contains. There is a heavy reliance on document types being correctly identified and containing correct data. This means both parties must understand the document types and correct data before interacting. Missing or incorrect information will terminate the conversation. There is no mention of the problems introduced by alternative outputs such as errors or help requests and the explosion of states these alternative paths can generate.

WSCL version 1.0 is a more recent proposal from Hewlett Packard. WSCL 1.0[14] is also based on the document exchange model with interactions and transitions. Five types of *interaction*: send, receive, send-receive, receive-send and empty are defined. Each interaction specifies which document types are exchanged between the service and its client. A conversation is modeled as a collection of interactions with the order of interactions specified by transitions.

The WSDL Message Exchange Patterns [18] largely mirror the WSCL interactions. There are 7 message exchange patterns identified, In-Only, Robust In-Only, In-Out, In-Optional-Out, Out-Only, Robust Out-Only, Out-In and Out-Optional-In. The robustness identified in the pattern name is the result of there being an optional fault message in response to the message sent (Robust Out-Only) or received (Robust In-Only). This makes them very similar to Out-Optional-In and In-Optional-Out, the difference being in the type of optional message (normal or fault). The similarities are inevitable, the message patterns describe the exchange of one or more content carrying messages in one or two directions with the possible addition of an optional fault message. Many interesting interactions will comprise more than two messages and this means the patterns would need to be composed into sequences or conversations. The composition, sequencing and/or possible overlapping of patterns is not addressed in the specification.

Guiding clients using WSDL service descriptions is proposed in [3]. In this work, clients are told by the service which of its operations can be called next. The interactions are still performed in the context of the WSDL description, so no guidance about the types of inputs the service requires can be given. The path of interaction is driven by the client deciding which of the operations will elicit the desired result.

The design of capability plans, in terms of what information is required to perform the capability, the specification of alternative sets of information and checking coverage and reachability is beyond the scope of this paper. There is however, a good body of published work in designing web applications [9, 25, 33]. This work approaches application design from various perspectives (organizational, data or user centric). Although it is directed more at web site design and providing context sensitive data and navigation options to human users, several of the methodologies and techniques described could generate effective capability plans.

[14] `www.w3.org/TR/wscl10/`

5 Conclusion

This work addressed the problem of ad hoc interaction between services that have no prior knowledge of one another. A mechanism was described that allows heterogeneous services to communicate in the pursuit of their goals. The mechanism includes an interaction language based on well understood communication primitives and a plan language that allows service providers to describe the information they need. It has been demonstrated how messages in the interaction language could be generated and interpreted by service providers and their clients and how the plan language would be used in a dialogue manager to collect a set of data from clients.

The interaction mechanism is based on well understood primitives that have a broad basis of support, it is easy to understand and can model simple or complex interactions with error handling and help. The mechanism is executable and will allow loosely coupled services to interact with one another at runtime without prior agreements in place. The language is structured to allow efficient, unambiguous and easy interpretation of messages.

Flexible and robust capability plans are built by offering alternative styles of input request (Input and Pick) and alternative sets of inputs when the client cannot satisfy the initial demand for input. Flexibility is also provided by allowing clients to request alternative names for parameters and datatypes to those used in the initial request.

Dynamic disambiguation of terminology is an important feature of this interaction mechanism. It is the way help is provided to ad-hoc interaction partners where there are no agreements in place on the syntax and semantics of the service's operations. A means of facilitating shared understanding between interaction partners is necessary to advance the vision of the semantic web [5] and the runtime interaction of loosely coupled heterogeneous services.

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

The interaction language provides performatives that give both providers and clients the means to control the dialogue in a cooperative manner at runtime.

This interaction mechanism supports loose coupling by not imposing requirements on the behaviour of clients beyond that clearly defined by the interaction language. The primary advantage of a loosely coupled solution to the problem of ad hoc interaction is that clients are not tied to specific service providers and implementations. This gives clients the flexibility to engage with any provider who can deliver the required functionality at runtime.

An implementation of the guide/dialogue manager in CPN (see appendix A) has demonstrated the interaction language and the plan language can be used together to interpret and generate messages and manage concurrent dialogues. The CPN implementation does not use or rely on proprietary technologies, and could be easily implemented in other programming languages.

A useful extension to the dialogue manager was suggested in section 3.2. Currently when a sub-dialogue is called it is assumed the capability is available locally. Extending

the guide with a service discovery mechanism could allow external service providers to be engaged to fulfil local sub-dialogue requests. In this way the service provider becomes the client of another service. This use of service providers by service providers shows how on-the-fly service composition is both achievable and useful.

Guided interaction provides the means to exchange information via conversational dialogue enabling web services and their clients to interact with one another in an ad hoc way at runtime without prior knowledge of (WSDL) service interfaces.

Acknowledgments The presentation of this research was funded by SAP Research Centre, Brisbane. Thanks to David Edmond and Marlon Dumas for their feedback on earlier drafts of this paper.

References

[1] Allen, J., D. Byron, M. Dzikovska, G. Ferguson, L. Galescu, and A. Stent: 2000, 'An Architecture for a Generic Dialogue Shell'. *NLENG: Natural Language Engineering, Cambridge University Press* **6**.

[2] Allen, J. F., G. Ferguson, and A. Stent: 2001, 'An architecture for more realistic conversational systems'. In: *Proceedings of the 6th international conference on Intelligent user interfaces*. pp. 1–8.

[3] Ardissono, L., A. Goy, and G. Petrone: 2003, 'Enabling conversations with Web Services'. In: *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*. pp. 819–826.

[4] Benatallah, B., F. Casati, F. Toumani, and R. Hamadi: 2003, 'Conceptual Modelling of Web Service Conversations'. In: *15th International Conference on Advanced Information Systems Engineering (CAiSE 2003), Proceedings*. Klagenfurt/Velden, Austria, pp. 449–467.

[5] Berners-Lee, T., J. Hendler, and O. Lassila: 2001, 'The Semantic Web'. *Scientific American*.

[6] Bollaert, J.: 2001, 'Crafting a wizard'. Available from IBM Developer Works: Usability. `www-106.ibm.com`, (25 September 2001).

[7] Bradshaw, J. M. (ed.): 1997, *Software Agents*. AAAI Press/The MIT Press, Menlo Park, California, USA.

[8] Burmeister, B., A. Haddadi, and K. Sundermeyer: 1993, 'Generic, Configurable, Cooperation Protocols for Multi-Agent Systems'. In: *From Reaction to Cognition - Fifth European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW-93, (LNAI Volume 957)*.

[9] Ceri, S., P. Fraternali, M. Matera, and A. Maurino: 201, 'Designing multi-role, collaborative Web sites with WebML: a conference management system case study'. In: *First International Workshop on Web-Oriented Software Technology, Proceedings*. Valencia, Spain. Available from: `www.dsic.upv.es/~west2001/iwwost01/files/contributions/StefanoCeri/Cer%iEtAl.pdf`, (20 January 2004).

[10] Cost, R. S., Y. Chen, T. Finin, Y. Labrou, and Y. Peng: 2000, 'Using Colored Petri Nets for Conversation Modeling'. In: F. Dignum and M. Greaves (eds.): *Lecture Notes in AI: Issues in Agent Communication*.

[11] Durfee, E. H.: 1999, 'Practically Coordinating'. *AI Magazine* pp. 99–115.

[12] Erbach, G.: 2001, 'Languages for the Annotation and Specification of Dialogues'. ESSLLI 01 Presentation, available from: `www.coli.uni-sb.de/~erbach/esslli01/index.html`, (12 June 2003).

[13] Foundation for Intelligent Physical Agents: 2000, 'FIPA Communicative Act Library Specification'. `www.fipa.org`, (9 March 2001).

[14] Franklin, S. and A. Graesser: 1996, 'Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents'. In: J. P. Muller, M. J. Wooldridge, and N. R. Jennings (eds.): *Intelligent Agents III Agent Theories, Architectures, and Languages. ECAI Workshop (ATAL) Proceedings*. Budapest, Hungary, pp. 21–35.

[15] Frolund, S. and K. Govindarajan: 2003, 'cl: A Language for Formally Defining Web Service Interactions'. Technical Report HPL-2003-208, HP Laboratories, Palo Alto.

[16] Gamma, E., R. Helm, R. Johnson, and J. Vlissides: 1995, *Design Patterns: Elements of Reuseable Object-Oriented Software*. Reading, Mass: Addison-Wesley Longman Inc.

[17] Gelernter, D.: 1991, *Mirror Worlds: Or the Day Software Puts the Universe in Shoebox...How it Will Happen and What it Will Mean*. New York: Oxford University Press.

[18] Gudgin, M., A. Lewis, and J. Schlimmer (eds): 2004, 'Web Services Description Language (WSDL) Version 2.0 Part 2: Message Exchange Patterns, W3C Working Draft.'. Available from:`www.w3.org/TR/wsdl20-patterns/`, (5 July 2004).

[19] Halpin, T.: 2001, *Information Modeling and Relational Databases: from conceptual analysis to logical design.* San Diego, CA, USA: Morgan Kaufmann Publishers.

[20] Hanson, J. E., P. Nandi, and S. Kumaran: 2002, 'Conversation support for Business Process Integration'. In: *Proceedings 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC-2002)*. pp. 65–74.

[21] Harel, D. and A. Naamad: 1996, 'The STATEMATE Semantics of Statecharts'. *ACM Transactions on Software Engineering and Methodology* **5**(4), 293–333.

[22] Hulstijn, J.: 1999, 'Modelling Usability: Development Methods for Dialogue Systems'. In: J. Alexandersson (ed.): *Proceedings of the IJCAI'99 Workshop on Knowledge and Reasoning in Practical Dialogue Systems*. Stockholm.

[23] Kuno, H. and M. Lemon: 2001, 'A Lightweight Dynamic Conversation Controller for E- Services'. Technical Report HPL-2001-25R1, Hewlett Packard Laboratories, Palo Alto.

[24] Labrou, Y. and T. Finin: 1998, 'Semantics for an agent communication language'. In: M. Wooldridge, J. Muller, and M. Tambe (eds.): *Agent Theories, Architectures and Languages IV.*, Lecture Notes in Artificial Intelligence. Springer Verlag.

[25] Martinez, A., J. Castro, O. Pastor, and H. Estrada: 2003, 'Closing the gap between Organizational Modeling and Information System Modeling'. In: *WER03 - VI Workshop em Engenharia de Requisitos, 2003*. Piracicaba, Brasil.

[26] Meandzija, B.: 1990, 'Integration through meta-communication'. In: *Proceedings of IEEE INFOCOM*. pp. 702–709.

[27] Neches, R., R. Fikes, T. Finin, T. Gruber, R. Patel, T. Senator, and W. R. Swartout: 1991, 'Enabling technology for knowledge sharing'. *AI Magazine* **12**(3).

[28] Oaks, P., A. ter Hofstede, and D. Edmond: 2003a, 'Capabilities: Describing What Services Can Do'. In: M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang (eds.): *First International Conference on Service Oriented Computing (ICSOC 2003), Proceedings*. Trento, Italy, pp. 1–16.

[29] Oaks, P., A. H. M. ter Hofstede, D. Edmond, and M. Spork: 2003b, 'Extending Conceptual Models for Web Based Applications'. In: G. Goos, J. Hartmanis, and J. van Leeuwen (eds.): *22nd International Conference on Conceptual Modeling (ER 2003), Proceedings*. Chicago, USA, pp. 216–231.

[30] Odell, J. J., H. V. D. Parunak, and B. Bauer: 2001, 'Representing Agent Interaction Protocols in UML'. In: P. Ciancarini and M. Wooldrige (eds.): *Agent-Oriented Software Engineering*. Springer-Verlag, pp. 121–140.

[31] Paurobally, S. and J. Cunningham: 2002, 'Achieving Common Interaction Protocols in Open Agent Environments'. In: *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS02)*. Bologna, Italy.

[32] Preece, J.: 1995, *Human Computer Interaction*. Reading, MA.: Addison-Wesley Publishing Company.

[33] Schwabe, D., L. Esmeraldo, G. Rossi, and F. Lyardet: 2001, 'Engineering Web Applications for Reuse'. *IEEE Multimedia* **8**(1), 20–31.

[34] Searle, J.: 1969, *Speech Acts: An Essay in the Philosophy of Language*. New York: Cambridge University Press.

[35] Toivonen, S. and H. Helin: 2003, 'Representing Interaction Protocols in DAML'. In: L. van Elst, V. Dignum, and A. Abecker (eds.): *Agent Mediated Knowledge Management International Symposium AMKM 2003, Stanford, CA, USA, March 24-26, 2003, Revised and Invited Papers*. pp. 310–321. Lecture Notes in Artificial Intelligence , Vol. 2926.

[36] Venkatraman, M. and M. P. Singh: 1999, 'Verifying Compliance with Commitment Protocols: Enabling Open Web-Based Multiagent Systems'. In: N. Jennings (ed.): *Autonomous Agents and Multi-Agent Systems*, Vol. 2. Kluwer Academic Publishers, pp. 217–236.

A CPN models of a guide and a generic client

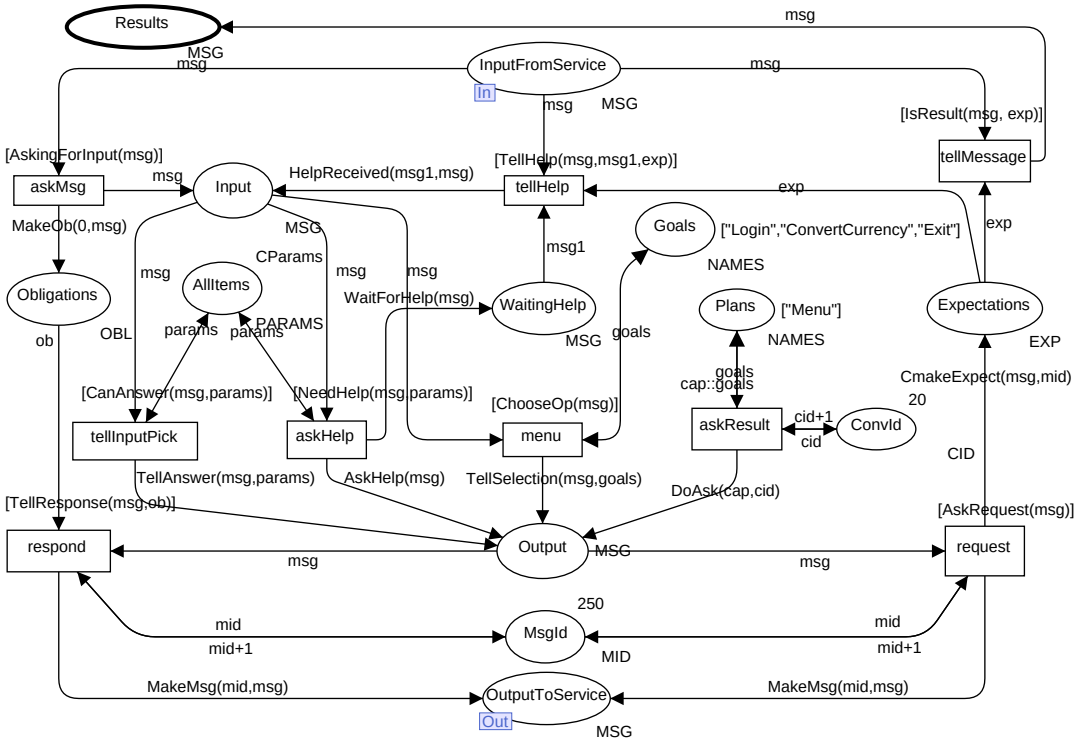

Figure 10: A service client

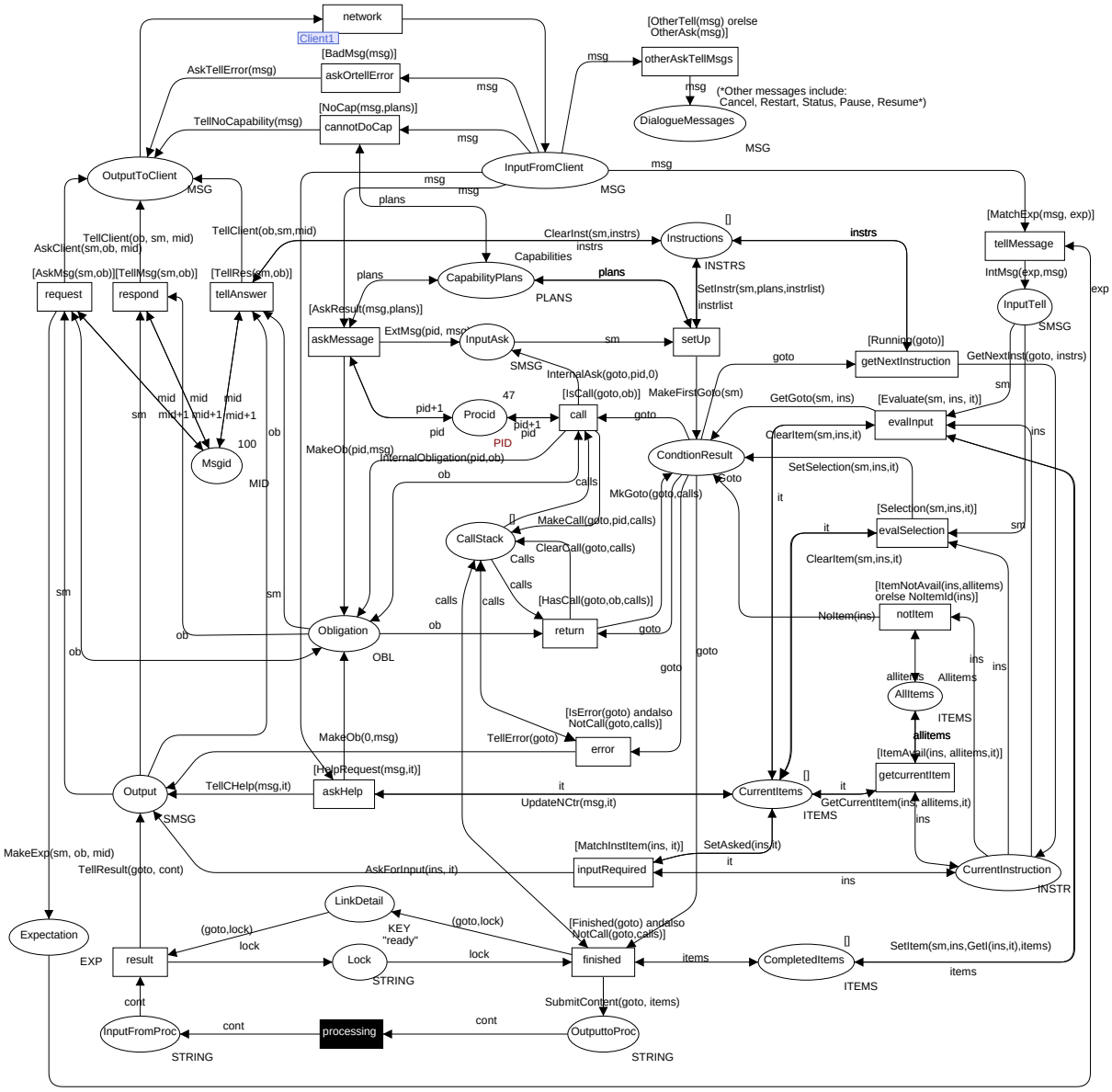

Figure 11: A guide represents the service provider