

5.5 Related work

IBM's Conversational Support for Web Services - CPXML [84, 85] describes Conversation Policies (CP)¹³. A conversation policy is 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 child states (similar to sub-dialogues) 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 realised by the specifications. Later work has reported using conversation policies to provide solutions to the perceived deficiencies in BPEL [97], such as the ability to maintain the state of message exchange sessions, to provide message adaptation, and manage nested protocols. An implementation of a conversation manager that runs a CP has not been reported.

The SELF-SERV platform is the implementation vehicle for the conceptual modeling of conversations in [14]. This mechanism also uses a state chart based representation. A conversation manager, implemented on the SELF-SERV platform, uses the state chart representation 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 that 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 terminology. Clients have no control over the management of the dialogue, such as requests to pause and resume etc. It is not suggested that the conversation model will need to be shared with clients, this implies that the service being modeled is reactive, in the sense that it reacts (in the specified manner) when clients elect to send it WSDL messages.

From Hewlett Packard [99] and [69] the Web Services Conversation Language (WSCL) models the conversation as the third party in an interaction. A conversation controller keeps track of conversation and changes state based on the types of messages, that is the type of the XML document the message contains. There is a heavy reliance on document types being correctly identified and containing correct data, (or vice-versa). 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.

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

WSCL version 1.0 is a more recent proposal from Hewlett Packard. WSCL 1.0¹⁴ 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 ordering of interactions specified by transitions.

The WSDL Message Exchange Patterns [77] 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 [100] and [133]. 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 as to 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.

5.6 Discussion

This chapter addressed the problem of ad hoc interaction between services that have no prior knowledge of one another. A mechanism was introduced that allows heterogeneous services to communicate with one another 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. Messages in the interaction language can be generated and interpreted by service providers and their clients and the plan language can be used in a dialogue manager to collect a set of input data from clients.

The interaction language and the plan language are both based on well understood primitives used routinely in the computing domain. The interaction language is structured to allow efficient, unambiguous and easy interpretation of messages.

¹⁴www.w3.org/TR/wsc110/

The plan language is based on a well understood binary choice control construct. It allows the definition of plans which are flexible and robust in the face of their clients diverse range of competency and access to information. The plans can be as simple or as complex as necessary to model any type of capability requiring input from clients.

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.

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 work. There is however, a good body of published work in designing web applications [35, 37, 36, 57, 78, 112, 137]. 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. Another approach [164] for describing composition plans uses a more expressive language than is available for capability plans but is pitched at a higher level than the one to one interaction dealt with here.

The dialogue mechanism as implemented asks separately for each input parameter value, this is appropriate behaviour with clients that have no prior knowledge of the service's requirements and when item specific help may be necessary. Some clients will have prior knowledge from the input signature definitions defined in the capability description. The implementation could optimise the collection of inputs for these clients by asking for input for several items in one message.

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 whether previously collected items are reused or collected anew.

A useful extension to the dialogue manager was suggested in section 5.3.2. Currently when a sub-dialogue is called the implementation assumes the capability is available locally. Extending the dialogue manager 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.

Another possible use for the plan language is by a data mediator in the selection and composition phase of service use. Using the plan language a data mediator can create a composition plan to orchestrate the selected services into a processing chain. In this scenario it becomes more important to incorporate the facility for

partial processing of results as the plan is enacted. For example, it may be necessary to process some data that has been received in order to pass the results onto the next service in the chain.

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 that allows focussed problem solving. Error information is also useful for reviewing operating performance and compliance checking.

The plan language supports traversing in order (sequence) and offers a conditional goto (choice) with conditions as expressions. There is no parallelism and function calls are non-recursive. The language is sufficiently complete for its purpose, which is to determine whether a client can provide at least one set of input data that can be used to deliver the functionality of a service represented by the guide.

Plans created with this language can be represented as directed graphs. This allows the use of techniques to detect cycles, which would uncover possible termination problems and to detect unreachable nodes which would uncover superfluous parts of plans. Formal analysis of plans though is outside of the scope of this work.

Dynamic disambiguation of terminology is an important feature of this interaction mechanism. It is the way help is provided to ad-hoc interaction partners that have no agreements in place on the syntax and semantics of the service's operations. A means of facilitating shared understanding between interaction partners at runtime is essential for loosely coupled heterogeneous services.

An extension that may be useful is to allow clients to clarify or seek help with regard to the results delivered by the guide. At present the dialogue terminates when the result is sent, preventing the client from seeking further information from the guide. There are several ways this extended assistance could be implemented.

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 that can deliver the required functionality at runtime.

The implementation of the guide/dialogue manager presented in section 5.4 demonstrated that the interaction language and the plan language can be used successfully in an ad hoc interaction context. The implementation does not use or rely on proprietary technologies, and could be easily implemented in other programming languages.

The guide is a kind of “command interpreter” this means it can run a plan for any capability. All that is required apart from the capability plan is a description of the items and parameters the plan refers to, and a means to access the back end functionality that provides the capability.

The interaction mechanism presented in this chapter satisfies all the evaluation criteria suggested in section 5.1. It is based on well understood primitives that have a broad basis of support, it is easy to use and can model simple or complex interactions that provide 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.

Future directions for the guide include combining the provider and client. In the implementation the provider activities were separated from the client activities. This was a conceptual convenience rather than a logical necessity. All of the computation contained in the client module could be easily integrated into the provider, giving dual provider-client services that communicate with one another to request and deliver capabilities.

