DYNAMIC USER INTERFACES FOR SEMI-STRUCTURED CONVERSATIONS

James E. Hanson, Prabir Nandi, Santhosh Kumaran
IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

Paul Foreman
IBM Software Group, Austin, TX, USA

Keywords: Software architecture and engineering, Computer-mediated communication, Interaction design

Abstract: The growing complexity of application-to-application interactions has motivated the development of an architectural model with first-class support for multi-step, stateful message exchanges—i.e., conversations—and a declarative means of specifying conversational protocols. In this paper, we extend this architectural model to encompass UI-enabled devices, thereby enabling it to cover human-to-application conversations as well. This permits either participant to be human-driven, automated, or anywhere in between, without affecting the nature of the interaction or of the other participant. The UI-enabled conversational model also reduces the difficulty of developing conversational applications, providing significant benefits both for UI and for application developers. We describe the architecture of a UI-enabled conversational system that supports a variety of user devices, and includes a means by which UI markup may be automatically generated from the conversational protocols used. We go through a sample application currently implemented using a commercially available application server, and further describe a graphical tool for editing and testing conversational protocols, that significantly eases the protocol development process.

1 INTRODUCTION

Increasingly, interactions between servers and clients in the World Wide Web are taking the form of *conversations*—i.e., multi-step, stateful, bilateral (or multi-lateral), correlated sequences of messages (for example, consider any mature e-commerce storefront). The same is true of interactions between applications in which no human is involved; and the growth of Web Services seems likely to accelerate this trend. This in turn led us to propose in (Hanson et al., 2002) an explicit conversational model for application interaction.

The present paper builds on that approach, extending the model to cases in which one (or both) of the "applications" is a human operating a UI-enabled device such a browser, PDA, or Web phone. In doing so, we show that a single model of interactions can be used across an extremely wide variety of applications from human-facing to fully automated, as well as hybrid cases in between.

We will be taking full advantage of the *semi-structured* nature of conversations specified in the model. By this is meant the feature that conversations frequently follow common, reusable patterns which can be expressed in terms of formal structural constraints on message exchange, but the number and character of these patterns is not known *a priori*; in fact, the patterns are continually evolving.

It was argued in (Hanson et al., 2002) that the use of explicitly declared conversational structures is a key enabler for complex application-to-application interactions. As we will see, it is also a powerful tool for simplifying the creation of markup for complex application-to-human interactions. Thus an additional benefit of the conversational model of interactions, and the architecture and programming model that it implies, is that it simplifies the development and maintenance of interactive Web sites. Since patterns of message exchange are first-class entities, they can be added, modified, replaced, etc., as a single unit.

E. Hanson J., Nandi P., Kumaran S. and Foreman P. (2005).
DYNAMIC USER INTERFACES FOR SEMI-STRUCTURED CONVERSATIONS.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 48-58
DOI: 10.5220/0002519100480058

In the next section, we review the conversational model in which the UI is operating. Section 3 then turns to the UI architecture itself. In section 4, we describe an example scenario we have implemented. Subsequent sections draw relationships to other work and conclude with a sketch of future work.

2 CONVERSATION MODEL

In this section, we briefly review the conversational model of interacting applications. More detail may be found in (Hanson et al., 2002).

2.1 Summary of the Architecture

The conversational model is a high-level architecture appropriate for applications that carry on multi-step, stateful interactions with other applications. As shown in Figure 1, it consists in essence of connected subsystems: a messaging endpoint, a conversation management subsystem, and decision logic. The messaging endpoint supports the sending and receiving of messages using one or another messaging protocol. The conversation management subsystem takes care of conversational session information and state information, and the decision logic performs the remainder of the processing necessary to drive the application through the conversation. The only constraints we make on the messaging endpoint and the decision logic are that they must be able to interact with the conversation management system in the specified way; beyond that, they may be anything.

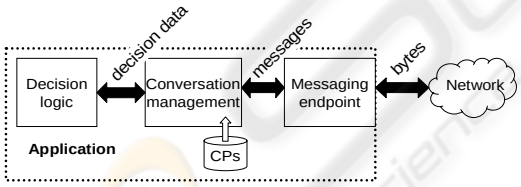

Figure 1: High-level view of the application architecture embodying the conversational model, (Only one of the conversing applications is shown.)

The conversation management subsystem is at base an engine for executing conversational protocols (cf. below). Each inbound message received at the messaging endpoint is passed into the conversation management subsystem. There, the conversation to which it applies is identified, it is validated against the currently-executing conversational protocol for that conversation, the protocol's state is updated as appropriate, and the relevant data is unpacked from the message. The new conversational state and the data unpacked from the message (which in Figure 1 are together referred to as "decision data") are then passed on to the decision logic for processing. For example, at some point in a bilateral negotiation, the decision data might give the conversational state as "counter-offer pending" and, as the data unpacked from the message, provide the contents of the counteroffer just received.

The inverse of this sequence is performed for outbound messages. The decision logic passes to the conversation management component an identifier of the decision made and any additional data associated with that decision (e.g., in response to the "counter-offer pending" input above, the decision logic might pass in "accept", or might pass in "make counter-offer" along with the contents of the new, outgoing counter-counter-offer). The conversation management subsystem validates these inputs from the decision logic against the conversational protocol, updates the conversational state, generates an appropriate message, and passes the message on to the messaging endpoint for delivery.

In this way, the decision logic is presented with an orderly sequence of validated inbound decision data that conforms to the conversational protocol in use (and, where an inbound message violates the protocol, it supplies a detailed context for error diagnosis and recovery); and it provides, for the decision logic developer, a rich, general-purpose interface for use in all conversations.

The notable feature of this architecture is that it separates the management of the interactions from the operation of the decision logic. This has a number of attractive properties, including:

- It factors out the conversation protocol support (which must be consistent between the conversing parties) from the agent's own internal business process (which will be different for each party).
- It permits an open-ended variety of decision-logic architectures to be plugged in. Thus the application developer can plug in virtually any kind of decision logic: e.g., workflow system, a software agent, or, as we describe in this paper, a person with a GUI. This is particularly attractive from the emerging Web Services/Service Oriented Architecture (SOA) perspective. Complex decision logic can be expressed as a service and discovered and plugged in dynamically to the conversation support system at runtime.
- It enables significant advances in conversation management, by means of meta-level conversational protocols (for conversations about the state of the conversation) to be treated in the same way as ordinary protocols. Thus reconnecting after a failure, resynchronizing, restarting after a pause, or handing a conversation off to a third party are all just

instances of conversational interactions, needing no special treatment.

- It promotes reuse of small, standardized messages as the bottom-level speech-act elements out of which complex interactions are assembled. (The conversational model does not of course prevent developers from taking a kitchen-sink approach, in which a single message carries with it a great deal of information that would more naturally be kept as conversational state; rather, it obviates the necessity for it.)

2.2 Conversational Protocols

Conversational protocols (CPs) are declarative specifications of message exchange sequences, giving, for example, schemas of messages that the conversing applications may send or receive at different points in the conversation. CPs are intentionally open-ended: that is, a given CP defines a set of messages that *may* be exchanged, without prescribing what messages *will* be exchanged. CPs as we use them here draw inspiration from the Pi-calculus (Milner, 1999) and from work in the software agents community (e.g., Greaves and Bradshaw, 1999).

One straightforward approach to CP specification is to express them as state machines, in which the states of the machine represent the different states of the conversation, and the transitions between states represent messages sent by one or the other participant (cpXML, 2002). This is the approach taken here.

Figure 2 shows an example of a conversational protocol, the "Haggle" CP, in the form of a state-chart diagram. In this CP, two participants (one playing role "A", the other "B") exchange offers and counteroffers until one or the other participant accepts the current offer, or cancels the negotiation. Execution of the CP begins in the state labelled "<<initial>>", and follows any sequence of the transitions until one of the states labelled "<<terminal>>" is reached. Transitions between states represent messages sent by one or the other participant, and are labelled "*sender-role: message-name*" (labels are in boxes). The different message-names are shorthand for detailed message schema information, such as WSDL portType and operation names (W3C, 2001) or URIs of XML Schema instances, in the CP itself. The Haggle CP is used in the scenario described in Section 4.

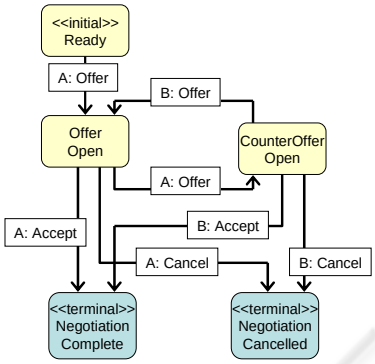

Figure 2: The Haggle CP.

Another key feature of cpXML, also illustrated in Section 4, is nestability—i.e., the ability to assemble complex protocols out of other, simpler, separately specified protocols. This is done by means of a special state type, the "in-child" state. When a CP enters an in-child state, the current state of the current CP is saved while another CP is loaded and executed. The "parent" CP forms the context in which the messages exchanged in the "child" CP are interpreted.

3 DECISION LOGIC ARCHITECTURE

The conversation support architecture extends naturally to encompass human users participating on one side of the conversation. Instead of being processed automatically (e.g., by a node in a workflow), the outputs of the conversation management component are transformed into *markup* for rendering on a human computing device. Similarly, the user's decisions (and attendant data) are extracted from the user device and transformed into decision-data inputs to the conversation management subsystem. (Note that this seemingly inverts the usual way of thinking about user devices: we are treating them as part of the "back end" decision logic subsystem.) The overall architecture is as shown in Figure 3.

Inbound decision-data (from the conversation management system) is transformed into appropriate markup (HTML, WML, VoiceML etc.) for rendering on the user-chosen device. Prompts at decision points defined by CPs are transformed in the same way. User supplied data is likewise transformed into a message format readable by the conversation support system.

The UI Controller provides the ability to plug-in markup specific translation engines. When the

conversation session is first established, the user device is automatically detected and the appropriate translation engine is selected. In addition the corresponding device-specific *presentation policy* is retrieved from the presentation policy repository. The presentation policies contain mapping rules to transform between the message schemas used in a given CP and the particular markup to be used to deal with those schemas.

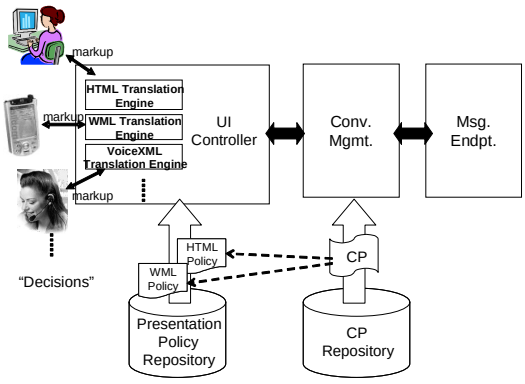

Figure 3: High-level architecture of the UI-enabled decision logic.

Separating the rendering of content (presentation policies) from its production (conversational protocols) allows the capability of extending support for additional devices simply by using the device specific presentation policy.

The structure defined by a CP, as executed by the conversation management subsystem, is invaluable for the generation of the markup. The current state of the CP gives the current conversational context. That context determines whether information is to be gathered from the user, and, if so, what the nature of that information is. Thus, for example, when the conversation enters a state in which the user may or must send a message, the transitions from that state give the names of the alternative messages that may be sent, and the schema associated with each transition gives the message-content that must be supplied by the user. It is straightforward to map the set of alternative message-names to a UI list-selection element; and possible (though not always aesthetically pleasing) to map the message-contents to a form to be filled out.

3.1 Presentation Policies

As discussed in the previous section, Presentation Policies contain the logic of translating between CP messages and its device specific markup. This approach is made practical by the semi-structured nature of the conversation, That is, for any given CP, there is a predefined and—usually—relatively small set of messages for which a presentation policy is needed to supply transformation information. It also helps that the use of CPs encourages the use of simple, reusable messages, for which the same transformation information may be reused.

There are 3 types of translation encoded in a presentation policy:

- mapping *received* messages to the markup
- markup to generate *form screens* to gather user data
- mapping and transforming *user data* from the screens to the message formats defined in the CPs

The logic of using screen real-estate optimally is a key function of the presentation policies, second of course, to its message translation chores. But a key feature of the system is the ability for the UI to operate in cases where no special presentation policy is available. This means that the minimum information required to carry on a human-to-application conversation is the CP or CPs used by the application—which are likely to be available from the application itself. In our current implementation, the default UI is automatically generated in the translation engines, which provide a default mapping if a presentation policy is unspecified. This default mapping is workable, if somewhat deficient in aesthetics and convenience, especially in cases where the messages are relatively small and simple. Further sophistication, for example, to enforce corporate branding, guaranteeing a consistent look-and-feel, etc., is accomplished by overriding the default rendering.

Just like CPs, presentation policies may be downloaded anytime, from anywhere. Conversation partners may supply their own PPs for look-and-feel customization, but user can also reuse the same PPs across different partners.

3.2 Architecture Variations

The architecture illustrated in the previous section can be adopted in a variety of ways.

Thin Client. In this case the conversation support and the UI Controller live on a server, with which the user devices communicate in the usual way. The server might, for example, be operated by an ISP, and conversation-support services might be part of its value-add for customers. The ISP would take care of downloading CPs and PPs as needed, as well as managing conversational state for its users, as shown in Figure 4.

Users subscribe to services offered by the ISP and interact via ordinary web browsers. ISP's can maintain preferred PP's as part of the user's profile. The ISP can also embed the conversation-UI as a frame in a partner-supplied page. This would be appealing to retail partners, since they could reuse their storefront & navigation templates. The served page could support automatic updates using JavaScript with META_REFRESH (refresh enabled in an invisible frame) or invisible applet with live connect etc. Alternatively, sophisticated pub-sub infrastructure can be employed to push received messages to the user.

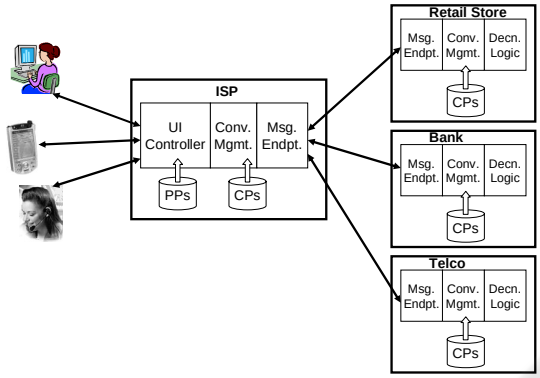

Figure 4: Thin client. Connections between the ISP and the providers' messaging endpoints on the right, and between the ISP and the user devices on the left, are over the network.

Thick Client. In this configuration, conversation support and the UI Controller are implemented as plug-in on the user device. Unlike the thin client configuration, the UI Controller is configured with a single markup translation engine: for example, simple Web browsers only require support for HTML markup translation. Thick clients seem more practical in managed environments where the necessary plugins can be effectively distributed on all user devices. Users download CPs and PPs of their choice from repositories maintained by their enterprise, or from individual department web sites. Users on the WWW, similarly, can download CPs and PPs from anywhere and have the freedom to do business with services providers of their choice. Intelligence can be added to the UI Controller, making it an agent rather than just a UI-generator. e.g., pre-fill mailing address. This permits single-point change (to mailing address) regardless of conversational partner. The thick client model, introduces a paradigm shift of sorts, moving transaction power closer to the user and making it possible for them to influence the way e-business transactions are organized and delivered, though at the cost of requiring more complex functionality at the user's end.

Collapsed Conversation Management. It is also of course possible for the conversation management to be hosted by the other party—e.g., the bank, Telco, etc., with which the user is conversing. This can be done in the obvious way, in which the user's conversation management system is connected directly to that of the service provider (thereby cutting out the intermediate step of messaging); or, alternatively, it is straightforward to implement a hybrid conversation management system that can simultaneously play both roles—i.e., that exchanges decision data with both sides of the conversation, foregoing the messaging endpoints entirely. This architecture is possibly closer to current practice than either the thin client or thick client (cf. Section 5). Because of that, it may be the easiest architecture to deploy initially. Note, however, that it realizes none of the advantages of reusability and user-customizability that the thin or thick clients provide.

Hybrid human-agent systems. There is a large body of research on software agents that specialize in user assistance, for which the natural place is as part of a hybrid decision logic component containing, as its human-facing part, the UI Controller. For example, (Hanson and Milosevic, 2003) discusses a hybrid system for CP-governed contract negotiation that contains automated validation of the contract's formulation. Other simple enhancements include pre-filling recognized fields in forms with values taken from a database of user preferences or from a history of previous user input; just-in-time search and download of CPs and PPs; etc. Because the ISP is inherently motivated to provide value-add services to its subscribers, the thin-client implementation in particular would seem to offer a promising path for the evolution of such enhancements. Furthermore, since the thin client and thick client implementations are under the control of the user (as opposed to being provided by the other party in the conversation), there is significantly less concern over privacy of user information.

4 PROTOTYPE IMPLEMENTATION

In this section, we describe a prototype implementation of the UI architecture, into which an example application has been implemented: obtaining a home loan.

The prototype uses a modified version of the Thin Client implementation, as shown in Figure 5. The Conversation Support for Web Services toolkit

(CS-WS, 2002), available from IBM alphaWorks, provides the conversation management runtime. It uses two machines for the two sides of the conversation, each running IBM's WebSphere Application Server (WebSphere, 2004). The two vertical lines represent firewalls. One of the machines is deployed in the company's DMZ (De-militarized Zone), processing the customer side of the conversation, while the other is deployed in the intranet connecting the mortgage application's business logic. The UI Controller is implemented as a servlet and deployed as a web application on the customer-side server. Customers use simple Web Browsers and HTTP to communicate with the servlet engine. Connectivity between the two conversation support machines is achieved with Web Services protocols.

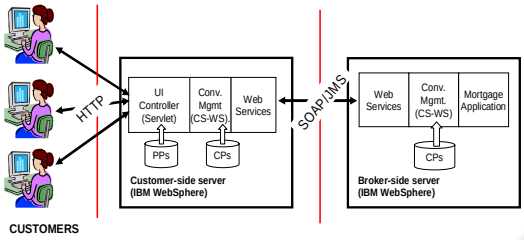

Figure 5: Deployment of the sample application

4.1 The scenario

A customer wants to obtain a loan for the purchase of a new home. The customer initiates a conversation with a mortgage broker, and immediately sees a form to fill out, with fields for name, address, etc.

What has happened is that, upon establishing the conversation, the conversation managers for both the customer and the broker loaded the first CP they will use.

Figure 6 shows the set of CPs that will come into play in the scenario and their parent-child relationships; the CPs themselves will be described in detail as we encounter them. The Mortgage Loan CP is the protocol describing the entire process. It makes use of the Qualify CP to establish whether the customer qualifies for a loan, and the Best Rate CP to settle on the parameters of the loan (here represented by only two parameters, the term and the interest rate). The Best Rate CP, in turn, makes use of the Negotiate Rate CP to settle on the interest rate; and the Negotiate Rate CP uses the Haggle CP of Figure 2 for the actual exchange of offers.

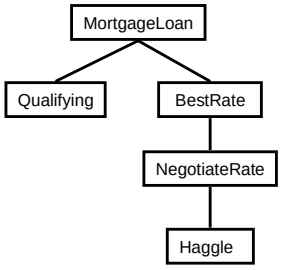

Figure 6: Conversational protocols used in the scenario.

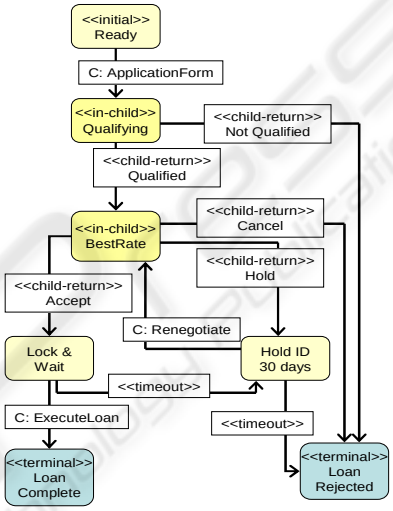

Figure 7: The MortgageLoan CP

In Figure 7 we see the MortgageLoan CP. The two roles for this and the following CPs are B and C, which are always played by the broker and the customer, respectively. Note also the two states labelled "<<in-child>>": when the conversation reaches these states, the given child CP is loaded and executed, while the parent waits. When the execution of the child CP is done, the parent CP is reactivated, and, based on the terminal state reached in the child CP, the appropriate "<<child-return>>" transition is taken.

As Figure 7 shows, the transition from the Mortgage Loan's initial state corresponds to an ApplicationForm message sent by the consumer. In our scenario, this has caused the customer-side conversation manager to send a decision request to its UI controller, which in turn has generated the form and sent it to the customer's browser.

The customer submits the form to the customer-side server, which (via the UI controller) translates it into decision data and submits it to its conversation manager. The conversation manager then updates the CP's state to Qualifying, which in turn causes it

to load the Qualifying CP, shown in Figure 8, as a child CP.

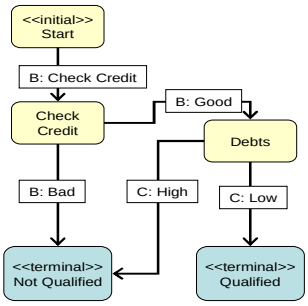

Figure 8: The Qualifying CP, version 1.

As specified, the broker sends a message to the consumer, indicating that a credit check is in progress. Note that the credit check itself is nowhere evident in the CP. Thus, the way in which the broker does the credit check (or even whether the broker does it at all) is invisible to the customer, as it should be.

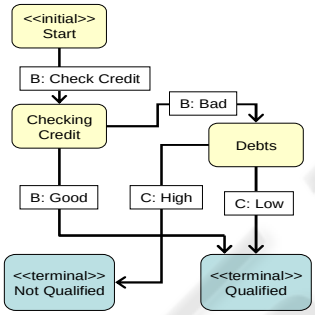

Figure 9: The Qualifying CP, version 2.

Having obtained the results of the credit check, the broker communicates them to the consumer by means sending either “Good” or “Bad” message. The consumer-side server converts this to markup for the customer’s browser. As shown in Figure 8, the “Good” message also triggers a query for the customer to answer: is the customer’s debt high or low? As per the presentation policy associated with the Qualifying CP, this shows up in the browser as a question, followed by a user-input area pre-filled with the options “High” and “Low”. The user selects one of the options, which drives the CP to one of its terminal states.

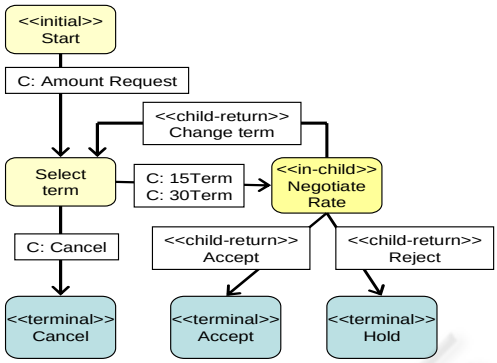

Figure 10: The BestRate CP.

One of the advantages of having a separate CP for the qualification phase is that it can be easily replaced with a new version. E.g., suppose that the broker decides to replace the Qualifying CP of Figure 8 with the version shown in Figure 9. For each server, this can be done simply by replacing the CP file; or, alternatively, by modifying, in the parent CP (Mortgage Loan), the URI for the child CP to load in the “Qualifying” state. The broker-side server can communicate this change to the customer-side server at any time prior to the execution of the Qualifying CP, for example by using a meta-level CP. Note also that the substitution is particularly simple in this case, since the messages used (and, therefore, the presentation policy needed to generate and to parse the markup) are the same for both versions.

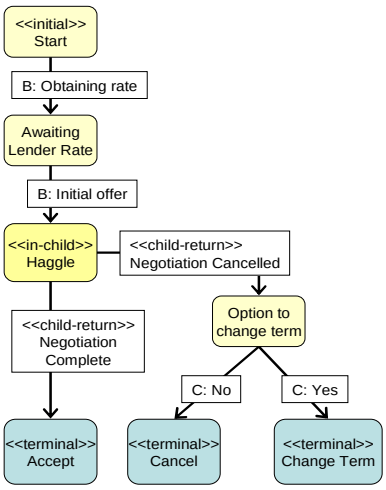

Figure 11: The NegotiateRate CP.

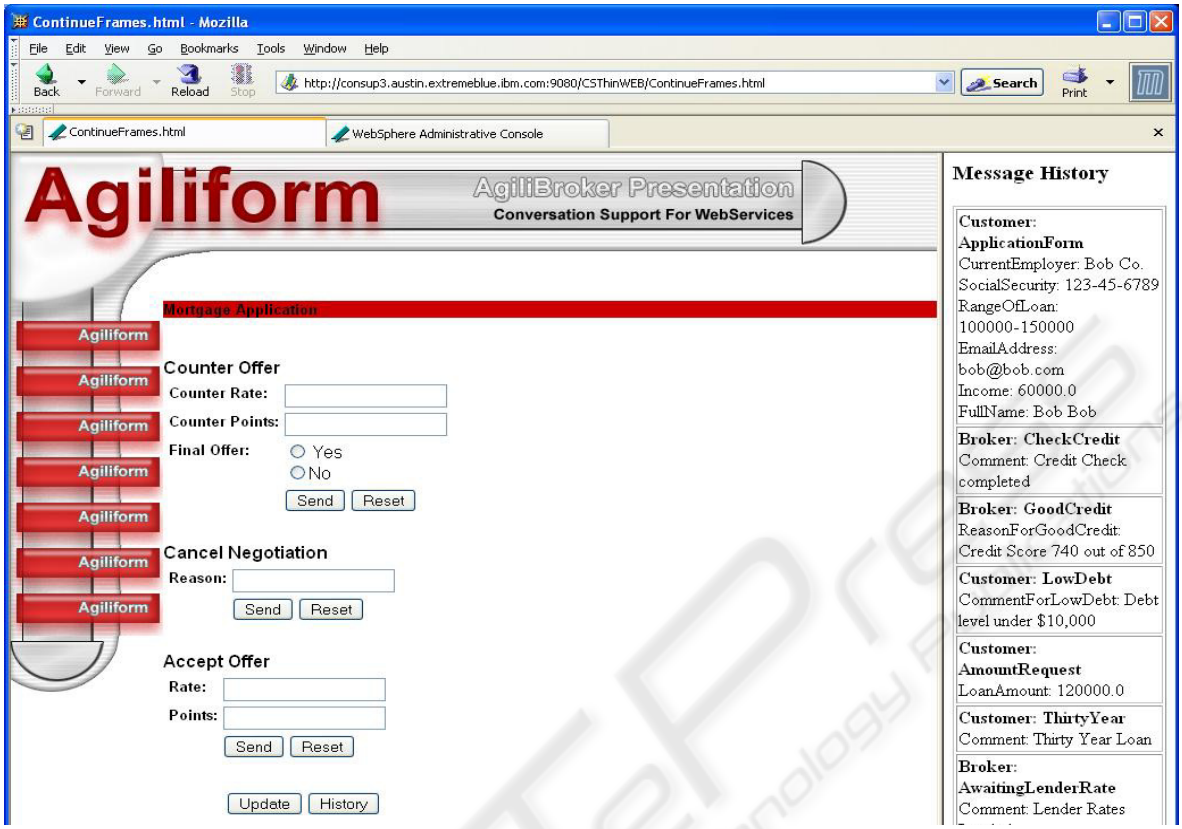

Figure 12: Screenshot of the application in action. The conversation is in state "Offer Open" of the Haggle CP (cf. Fig. 2), with the customer playing role B. The Message History column keeps a running commentary of messages exchanged.

Once the broker has decided to proceed with giving the customer a loan, they enter the BestRate CP, shown in Figure 10. In this CP, the customer names an amount for the loan, and then selects a term. (Note that a pair of parallel transitions labelled "C: 15Term" and "C: 30Term" are shown in a single box with a single arrow.) Once the term is selected, the broker and customer load the NegotiateRate CP, shown in Figure 11. This CP, in turn, loads the Haggle CP of Figure 2.

At the same time as it loads the NegotiateRate CP, the broker starts a backend process that negotiates with several lenders to determine the best rate available for that term. This conversation happens on the side and the customer is never aware of it; all the customer sees is a message saying that the broker is obtaining a rate. This gives the broker more bargaining power by hiding the lender negotiations from the customer. Each lender enters into a negotiation with the broker using the Haggle CP to make offers and counter offers over rates. Once the best rate is collected, the broker resumes the conversation with the customer and provides a rate.

At this point, the customer and broker enter into a negotiation using the Haggle CP, with the customer playing role A and the broker playing role B. In our scenario, the customer and broker at this point do not come to an agreement on rate and the conversation is suspended while the customer seeks other offers.

When the customer suspends the conversation, the customer-side server stores the conversation's state information, including the conversation IDs, the CPs in use, the current state and role played by the consumer of each CP, and the history of the messages exchanged. (In the present implementation, the broker does not suspend its side of the conversation, but leaves it "live"; however there is no impediment to having the broker suspend the conversation as well.) Later, when the customer reconnects with the customer-side server, he is offered the option of resuming a suspended conversation or starting a new one. If the customer chooses the first option, the stored conversation state information is reloaded into the conversation management subsystem, and the customer and broker resume where they left off.

The customer returns to the conversation after two days have passed and renegotiates with the broker. This time the customer logs in and sees that the Best Rate CP has been modified: in parallel with the transitions for 15 and 30 year terms, there is a new transition labelled "C: 20 Term", corresponding to a 20 year term. The customer decides to select this new term and restarts the negotiation. The broker has also added a new lender to his list with very competitive rates and this lender is able to come up with a rate that allows the broker to present an offer that the customer accepts. The customer signs a lock with the broker for 30 days and leaves the conversation.

4.2 The User Experience

Figure 12 shows a screenshot of the application in action, at the point in the scenario when the broker and customer are negotiating the details of the loan. The screenshot shows what would be displayed in the customer's browser. The conversation is in the Haggle CP, in the "B's Offer Pending" state—i.e., it is the customer's turn either to make a new counteroffer, to cancel the negotiation, or to accept the broker's offer. These three options, as defined by the transitions from the current CP state, are presented to the customer as three exclusive options. The customer must decide which option to choose, fill in the data associated with that choice, and click "Send". For example, to make a counteroffer, the customer enters values for "Counter Rate" and "Counter Points", selects the appropriate button to indicate whether he wants to assert that this is his final offer, and then clicks the "Send" button immediately below.

This figure illustrates the user's experience throughout the conversation. Each time the conversation reaches a decision point for the consumer—that is, each time the consumer has the opportunity for sending a message—the UI controller running on the customer-side server generates markup similar to that shown in Figure 12, in which the options open to the customer, which are determined by the current state of the CP being executed, are presented in the form of a set of mutually exclusive options, optionally with additional parameters. The user determines which option he wishes to choose, fills in the parameters and clicks the "Send" button for that option.

Running down the right side of Figure 12 is the history of the conversation. This is automatically updated every time the state changes. In each box is the name of the message that was sent (as given by the CP), and the values of the associated parameters. Note that the history shows the messages sent by both the customer and the broker.

The message history and the user input area are framed by the Agiliform logo and a simulated navigation bar. Agiliform is the name for this project, chosen by the IBM "ExtremeBlue" team that made it a reality. In practice, a vendor or ISP would supply markup for these areas, thus preserving the vendor's brand identity, navigation, etc.

In an earlier prototype, we also implemented a real-time graphical display of the stack of CPs currently in use (from parent to child to grandchild, etc.), and the state-transition graphs of each CP, with the current state highlighted. This was extremely useful in debugging, but was felt to be too complex to expose to the average user.

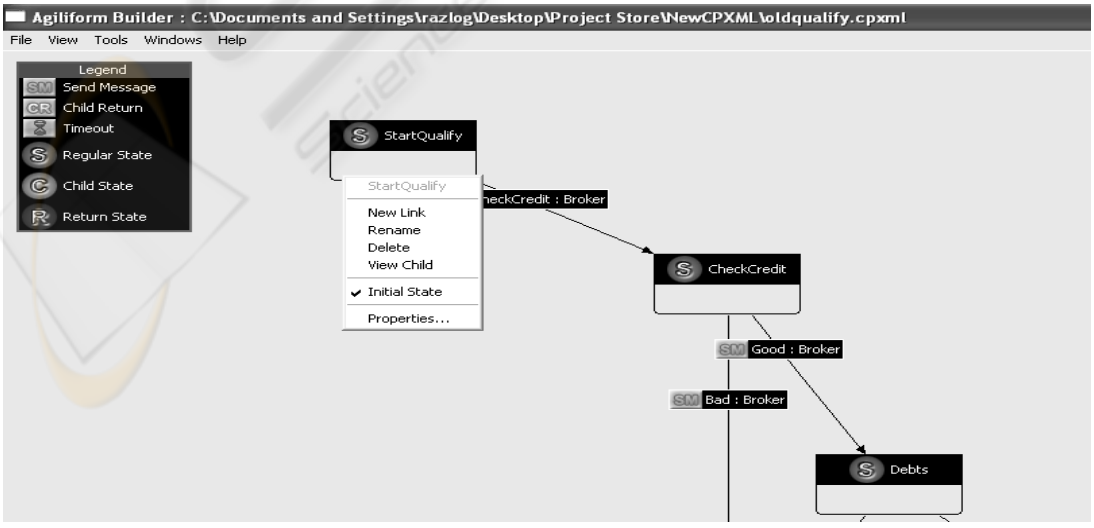

Figure 13: Screenshot of the CP authoring tool. The Qualify CP is being edited

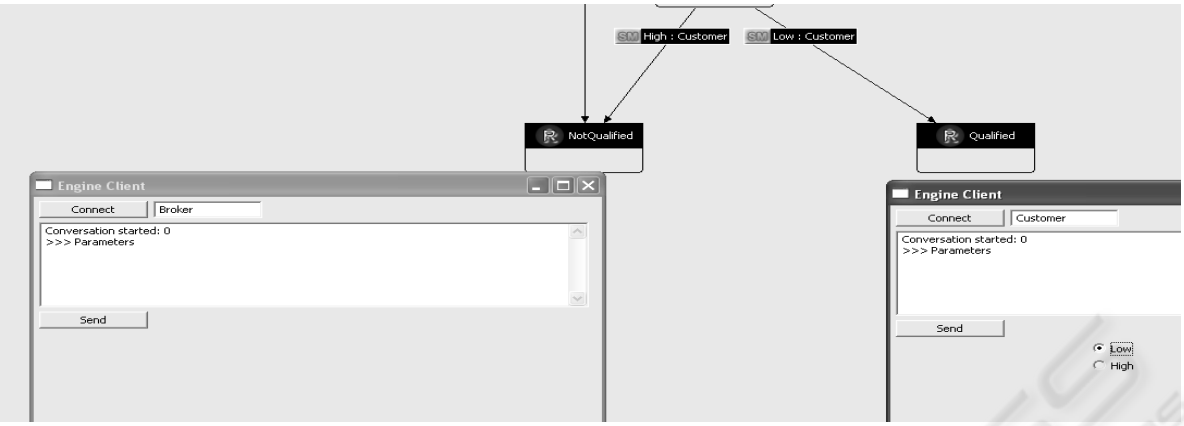

Figure 14: Screenshot of the CP authoring tool during a test of the Qualify CP. On the left is the window for the broker; on the right is that for the customer.

4.3 CP Authoring and Testing

In addition to the sample application, the Agiliform team created a CP authoring and testing tool. With the tool, protocol developers may create and modify CPs graphically by drawing their state-transition diagrams and filling in the necessary parameters on the states and transitions. The tool both reads and generates cpXML files; the CPs so generated are ready for deployment on the server. Figure 13 shows a portion of a screenshot of the authoring tool, with the Qualify CP loaded.

The authoring tool also has a testing facility. This permits the CP developer to manually play both sides of the conversation, and watch as the decisions of both parties exercised the CP through its states. A screenshot of such a test is shown in Figure 14. Here, the Qualify CP is being tested, and the conversation is at the point where the customer must specify whether he has a low or high amount of debt.

5 RELATED WORK

Most of the conversation-related work in the UI field is concerned with natural language processing and speech recognition techniques embedded in the UI. Thus a "conversational UI" is frequently understood to be a voice-activated UI, providing a mapping between the human voice and text (W3C, 1998). This work forms the core technology on which one of the translation engines plugged into the UI controller is based (cf. Section 3.1) and therefore is a powerful enhancement to the system we discussed in this paper; but is of course not to be confused with the notion of conversation used here.

The Struts framework (Apache.org, 2004) uses an XML configuration file to specify page sequences and the data shared and/or transferred across pages. There are also some browser-based products that drive interactions with a user using a decision script (Vanguard, 2003). Their applicability is primarily in the area of providing customer support. The script runs on the server and guides the customer through the troubleshooting process. These are in some important respects similar to the Collapsed Conversation Management architecture described in Section 3.2.

There is an extensive body of work on model-based UI development in which a high level specification language is used to describe the interface design. This specification would then be automatically or semi automatically translated to platform specific executable code or interpreted at runtime to generate the appropriate interface. This work may be usefully applied to improve the quality of the automatically generated UIs used in cases where a presentation policy has not been specified. A comprehensive overview of the architectural elements and their evolution within the UI community is detailed in (Szekely, 1996). Conversational protocols could provide the *task* and/or the *domain* model required for such tools and create highly sophisticated *presentation policies.*

6 CONCLUSION AND FUTURE WORK

In this paper we have described a way in which humans operating UI-enabled devices can be integrated into a world of application-to-application conversations. This integration hinges on a model of

application interactions in which conversations follow semi-structured, nameable, explicitly described protocols. As a consequence of this integration, applications of all types (human-operated, wholly automated, hybrid) may coexist as peers in the same overall system.

This simplifies the job of the back-end decision logic developer, since he can focus on the actual decision-making logic without being concerned with whether the user of the application is a human or another application.

It also simplifies the job of the UI programmer. In the CP authoring tool, we have seen an indication of the degree to which programming of complex interactions is simplified by the use of the conversational architecture. Multi-step stateful message exchanges can be designed with great ease using a simple graphical tool, tested immediately, and deployed effortlessly. The Conversation Protocol Builder can be easily integrated with one or other Model-based UI tools (e.g., Paterno, 2002) to automate the process of generating highly sophisticated , multi-modal UI to drive the human end of the conversation.

These two observations suggest that using the conversational model can significantly reduce the difficulty of developing even very complex Web applications, including both the UI and the back-end decision logic.

We close with a sketch of a possible direction for future work. Businesses providing manual user support typically tape interactions for later review. These audio records have proven to be extremely useful for training customer support personnel, determining customer and/or market demands, tracking customer relationships, etc. However, the cost and difficulty of extracting such information from the raw audio data puts it out of the reach of many small businesses. The conversation-aware user interfaces described in this paper, in addition to providing a viable means of automating user-facing operations, allows for easy storage, retrieval, and processing of past conversations. In addition to the messages themselves, conversation histories giving the paths that actual conversations take among the states of the CPs they use are particularly useful for this purpose. For example, in the mortgage loan application, the broker can use standard statistical tools to discover where (i.e., in which CP state) negotiations break down most frequently, and in what circumstances (e.g., is the rate too high?). Corrective measures can then be adopted. Developing innovative ways to store and analyze conversation data will be one of our major research directions for the future.

ACKNOWLEDGEMENTS

The authors would like to gratefully acknowledge the contributions of the IBM ExtremeBlue team in making the Agiliform application a reality: John Baggett, Anand Srinivasan, Philip E Light, Razvan Loghin, David Barnes, David P Greene, Ronald Woan, and Vishwanath Narayan. The authors also thank Yasodhar Patnaik and Terry Heath for their work on an earlier version of this paper.

REFERENCES

Hanson, J., Nandi, P., and Levine, D., 2002. Conversation-enabled Web Services for Agents and E-business, in *Proc. 3rd Intl. Conference on Internet Computing (IC-02)*, CSREA Press, pp.791-796.

Milner, R, 1999. *Communicating and Mobile Systems: the pi-calculus*, Cambridge Press, Cambridge, UK.

Greaves, M., Bradshaw, J. M., (eds.), 1999. *Proc. Autonomous Agents '99 Workshop on Specifying and Implementing Conversation Policies*.

cpXML, 2002. Conversation Policy XML, www.research.ibm.com/convsupport/papers/cpXML-v1.htm

W3C World Wide Web Consortium, *Web Services Description Language (WSDL) 1.1*, 2001. www.w3.org/TR/wsdl

Hanson, J., and Z. Milosevic, Z., 2003. Conversation-oriented Protocols for Contract Negotiations, in *Proc. 7th IEEE Intl.Enterprise Distributed Object Computing Conference (EDOC-2003)*, IEEE Press.

CS-WS, 2002. Conversation Support for Web services, www.alphaworks.ibm.com/tech/cs-ws

WebSphere Application Server homepage, IBM Corporation, 2004. www.ibm.com/software/webservers/appserv/was

The W3C Voice Browser Workshop www.w3.org/Voice/1998/Workshop/papers.html

Apache.org, 2004. Jakarta Struts Framework, jakarta.apache.org/struts/

Vanguard Corporation, Smart Servers, www.vanguardsw.com/DecisionScript/SmartServers.htm

Szekely, P., 1996. Retrospective and Challenges for Model-Based Interface Development, www.idi.ntnu.no/emner/tdt12/szekely-retrospective-CADUI96.pdf

Paterno, F. and Santoro, C., 2002. One Model, Many Interfaces, giove.cnuce.cnr.it/teresa/pdf/Paterno-CADUI2002.pdf