High Level, Multi-Agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example

R.J.A. Buhr[†], D. Amyot[†], M. Elammari[†], D. Quesnel[†], T. Gray[‡], S. Mankovski[‡]

Carleton University[†] and Mitel Corporation[‡], Ottawa, Canada

buhr@sce.carleton.ca, damyot@csi.uottawa.ca, {elammari, quesnel}@scs.carleton.ca, {tom_gray, serge_mankovski}@mitel.com

Abstract

This paper joins new concepts in software design with the BDI reference model for agents and with a novel way of developing executable prototypes. Within the resulting conceptual framework, high level prototypes of multi-agent systems flow in a systematic way from abstract descriptions of the emergent behaviour patterns we expect agents collectively to produce. The contributions of the paper are: a novel approach, centering around Use Case Maps (UCMs), for linking together behaviour requirements, software design models and BDI-style reference models; application of agent conflict resolution techniques to the practical problem of telephony feature interaction, following this approach; a way that scales up of resolving conflicts, using competing rule engines in the executable prototypes; and demonstration of the essence of an intended approach for future commercial systems.

1. Introduction

The work of this paper is motivated by the application of agents to telecom problems. Telecom systems provide a fertile field for agents. Technology and user requirements in this field are continually changing and agents offer the prospect of incrementally adapting to such changes. Telecom systems are distributed and agents offer new ways of dealing with distributed systems. Telecom systems continually have new features added to them, and conflicts between new and old features must be resolved, often dynamically (telephony feature interaction provides an example in this paper). Agents are very suitable for such problems because of their ability to handle conflict resolution situations dynamically. However, we see several issues in making agent systems practical for telephony applications, as will now be discussed.

Our experience is that the dynamic nature of agent solutions presents great difficulties for telecom software designers. Multi-agent systems tend to be dynamic on a large scale. For example, transient alliances among agents may be continually forming and dissolving as the system runs. Situations that are dynamic on such a large scale are not easy to represent or present for human consumption with software design techniques popular in the telecom community. In current practice, dynamic situations tend to get pushed down as detail in software design descriptions, or deferred to code. This has caused telecom software systems to be very resistant to the continual evolution that is required by competitive pressures. Changes and additions made without clear knowledge of how the dynamic situations of previous features were conceived can cause the entire system to become unstable. With current techniques, these situations can only be detected and corrected with extensive regression testing which prolongs development, delays time to market, and opens the field to competitors.

In contrast, agent models (e.g., BDI [20], COOL [1], and AOP [21]) handle system-wide dynamic situations in a high level, declarative fashion. Being declarative, however, such techniques do not offer a system picture for telecom software designers. Furthermore, they are agent-centric, meaning they do not provide descriptions at the level of systems of agents, but only of individual agents. Their agent-centric descriptions can be mapped to implementations of individual agents using conventional software design techniques, but the system behaviour is whatever emerges. Dynamic situations are seen only in terms of inter-agent messages and protocols, which is not adequate for visualization because it obscures the big picture under a cloud of details.

Visual techniques are well known to be needed for human understanding. Kendall [16], the DESIRE tool [11], and the Clearlake tool [15] model agent systems using object-oriented-style diagrams. Such techniques leave dynamic situations involving multiple agents to messages and protocols, which is not adequate for visualization, for the reasons mentioned in the previous paragraph (DESIRE raises the level of abstraction somewhat by including a meta-level of messages, but the point remains valid).

There are also other factors that make leaving everything to messages and protocols undesirable. In agent systems, protocols are secondary to the environmental conditions that agents encounter (resources available, diversity of actors affecting the system, and so on). The great variety of environmental conditions contributes to great complexity of conditions under which a particular protocol is to be used.

All these factors together obscure the system picture. This allows software designers to form completely incompatible views of the system picture, complicating and slowing down the developmental process.

Additional views of the two industrial authors of this paper and some of their colleagues on issues affecting the provision of telecom services with agents may be found in a contribution to PAAM97 [19] and also elsewhere [22][23][24][25].

To exploit the potential of agents effectively, telecom software designers need a way of bridging a large conceptual gap between the big picture, involving multiple agents, and software design details of individual agents. This way must be both easily understood by telecom software designers and easily related to BDI-style solutions for individual agents.

In the research described here, we satisfy these requirements with a technique called Use Case Maps (UCMs) [4][5][6][7][8][9]. We are developing a high level design and prototyping environment with UCMs at the top end, a rule-based prototyping environment connected to a telephone system simulator at the bottom end, and BDI-style models in the middle [10].

For clarity of exposition to an audience not familiar with UCMs, this paper develops these ideas by proceeding from the bottom up. Section 2 introduces the feature interaction example used in this paper. Section 3 describes how telephony feature interactions may be resolved in an environment that combines Java agents and competing CLIPS rule engines that resolve feature conflicts through a Micmac blackboard (this environment includes a MediaPath system that simulates actual telephony). Section 4 describes how the rules and other elements of this environment are derived in a systematic way from BDI-style descriptions. Section 5 describes how the BDI-style descriptions follow from path descriptions in UCMs. Section 6 discusses where we are going with this approach and how we think others might use it. Section 7 draws conclusions.

2. The Nature of the Feature Interaction Problem

We exercise these ideas in this paper on the problem of telephony feature interaction [3][12] because it is a conflict resolution problem that must be resolved dynamically, an appro-

priate type of problem for agents. The telephony feature interaction community has recognized the utility of agents for solving such problems [25].

Telephony feature interaction provides an example of a very general type of system problem, in which errors at the system level are created by inappropriate combinations of dynamically determined details in different parts of the system. Figure 1 illustrates an instance of such problem between two common features, namely Originating Call Screening (OCS) and Call Forwarding (CF). OCS forbids the establishment of a call to phone numbers on a screening list, while CF forwards incoming calls to another phone number. The two features interact inappropriately when *A*, whose OCS screening list includes *X*, calls *B*. Since *B*, who is subscribed to CF, is not available, the incoming call is forwarded to *X*. However, *X* was on *A*'s screening list, and therefore the connection should not have happened, as it violates assumptions related to OCS.

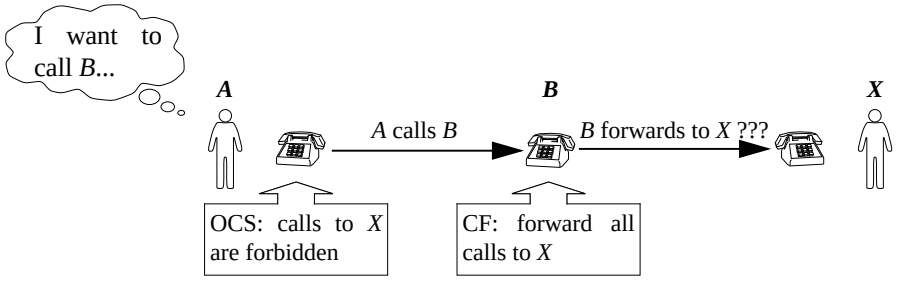

Figure 1 Illustration of a telephony feature interaction.

Do not be misled into thinking feature interaction is a simple problem by the fact that an example of feature interaction can be described in an understandable way so tersely. Telephony systems contain hundreds of features (not all are seen by users) and new ones are continually being added. Not only is there a combinatorial explosion of feature interaction possibilities, but the possibilities may be buried deep in complex software. Software complexity may be measured in tens of millions of lines of code.

Ways are needed of making the feature interaction problem more visible (this is one role of UCMs in this paper), of resolving feature interactions dynamically (this is the role of competing rule engines in our prototypes), and of enabling solutions to flow from high level descriptions (this is the another role of UCMs in this paper).

3. A Prototyping Environment, By Example

3.1 Prototype Environment Architecture

Our prototype environment is composed of several agents, which communicate with each other through a blackboard, and a Java interface to MediaPath [17], an open, standards-based communications server produced by Mitel. MediaPath is comprised of call control software and server telecommunication boards (voice processing board, trunk board, and line board).

Micmac [18] is a coordination tool that can be used by a multi-agent environment. In the Micmac system as developed, feature interactions are detected by the feature placing its intention to perform an action in the tuple space. Any other feature can then comment on the action and the originating feature can take this advice and decide how to proceed. In BDI terms, the feature is proposing an intention which other features can comment about. The agent is reasoning about the development of an intention.

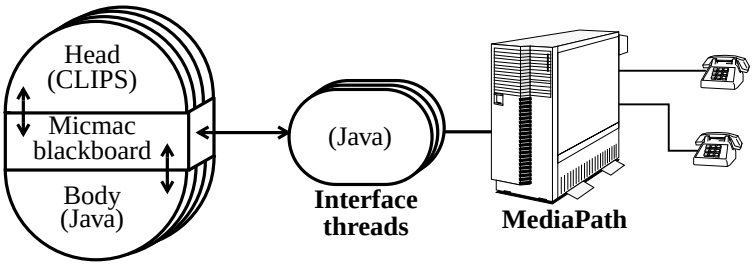

Figure 2 Coordination between agents and MediaPath through a Micmac blackboard.

As shown in Figure 2, each agent is composed of a head and a body. The head is where decisions are taken (plans are selected), and the body is where these decisions are carried out (plans are executed). The head is implemented declaratively using the CLIPS expert system [14], and the body is implemented in Java. The head and the body of an agent also communicate through the blackboard. This allows for the distribution of heads and bodies to different physical locations. We developed a Java interface for Micmac so that it can be used directly by the body half, and also by the head half whose CLIPS rules are interpreted by Jess [13], a Java implementation of CLIPS' essential features.

The core of Micmac is composed of a *tuple space*, which is an instance of a blackboard architecture. In such an architecture, entities (also called knowledge sources) communicate and invoke operations on the blackboard through a publish-subscribe mechanism. A *tuple* is a set of ordered pairs called *ingles*. Each ingle consists of a type (say `connectFrom`) and a value (say *A*). An example of a tuple is: `{ :connectFrom A :connectTo B :callID id }`. This would be a tuple that describes a call request from *A* to *B*. The tuple space enables coordination by allowing queries based on the matching of tuples by *anti-tuples*. An anti-tuple is a tuple that can be used as a query in the tuple space. In form, it is identical to a tuple except that the value of any or all fields may be replaced by a '`?`', which indicates a 'don't care' condition, similarly to a template. Tuple spaces are set up to match tuples with anti-tuples which agree in all fields except for the ones indicated by the '`?`' query, similarly to Prolog unification.

Four operations on the tuple space have been defined: *poke* places a tuple in the tuple space, *peek* queries the tuple space with an anti-tuple (matching tuples will remain in the tuple space), *pick* also queries the tuple space with an anti-tuple (but matching tuples will be removed from the tuple space), and *cancel* removes all matching anti-tuples from the tuple space. Durations for tuples can also be defined.

3.2 Competing Rule Engines Communicating via a Blackboard

Coordination of call processing applications with tuple spaces is accomplished by use of a permission/rejection mechanism among features. Features, such as the ones shown in Figure 3 (ORIGINATING, OCS, TERMINATING, and CF), are instantiated as discrete entities which will interact through the tuple space. Agent heads implement these features as competing CLIPS engines. This approach based on a blackboard simplifies the interaction protocols for the collaboration among agents, and it minimizes the need for maintenance when we add, modify, or delete features, even at run-time. This is especially true in the type of telephony applications that interest us, where no deterministic or optimal solution is known, and where opportunistic problem solving seems to be the most practical approach. This solution fits well with both our application domain and the agent paradigm.

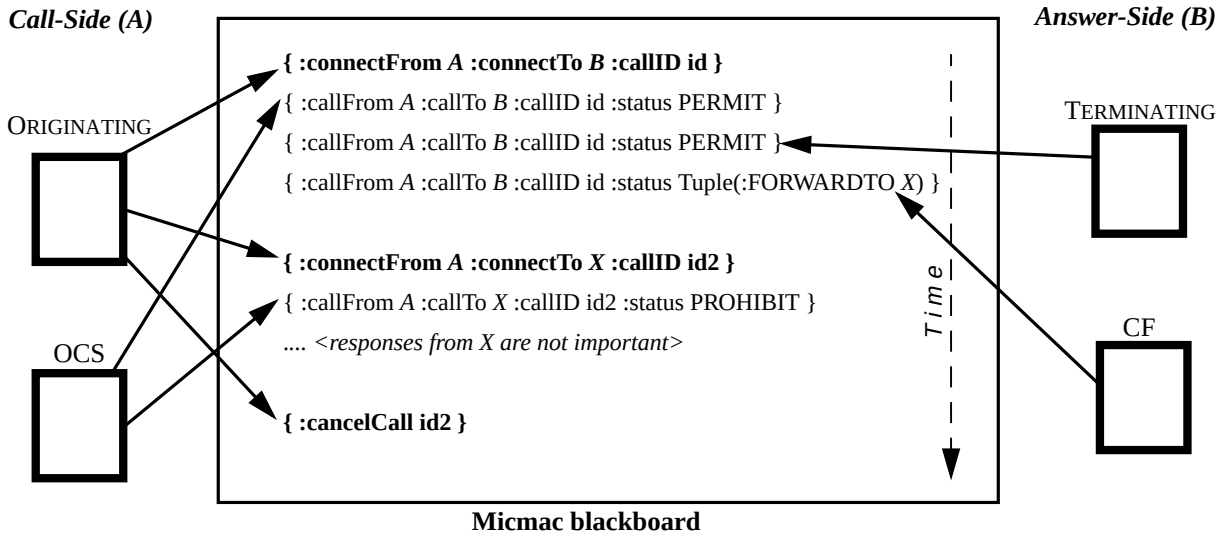

Figure 3 Competing rule engines communicating via a blackboard.

Figure 3 illustrates the feature interaction solution between OCS and CF in terms of messages exchanged in the tuple space. In this figure, messages are posted from top to bottom, as time passes. The arrows show the messages sent by the agents at some point in time (messages in bold are from the default ORIGINATING feature).

First, the ORIGINATING feature expresses its intention of connecting *A* with *B* by posting a proposal (or poking a tuple). It then waits, for a certain amount of time, for comments from other features. As comments can arrive in any order, this particular sequence is only one of the many possible global scenarios. Other features, which received the message (peeked using an anti-tuple), reply according to their internal set of rules. In our case, OCS and TERMINATING permit the call between *A* and *B*, but CF indicates that this call has to be forwarded to *X*.

The ORIGINATING feature decides its next action based on the new facts (beliefs) available. Many strategies can be defined for this decision mechanism, according to functional or business logic. In our case, we used the notion of local *salience*, which provides a priority to the rules associated to answers received from other concurrent features. Each status received corresponds to a specific salience within ORIGINATING. From the highest priority to the lowest, we have: PROHIBIT, REJECT, FORWARDTO, and PERMIT. The ORIGINATING feature reasons about the comments by sorting the corresponding rules according to their salience, and uses the one with the highest priority to decide its next move. A CLIPS engine always chooses the highest priority rule on its agenda. The other facts are then simply retracted from the local fact-list in order to avoid firing another rule based on those remaining facts. This approach scales up as many features can coexist and comment on proposals using such status, without the need for ORIGINATING to know how many other features are active.

The second proposal posted by ORIGINATING is therefore a connection between *A* and *X*, as suggested by the 'FORWARDTO *X*' tuple received from CF. As soon as OCS peeks at this proposal, it replies with a PROHIBIT status as *X* is in *A*'s screening list. From thereon, no matter the comments from the other feature agents (remember that PROHIBIT has the highest priority), ORIGINATING will cancel the call.

This scenario illustrates the resolution of conflicts between features that are active simultaneously. Feature interactions can usually be of three kind: violation of assumptions, indeterminacy, and data violation. The OCS/CF interaction introduced in Figure 1 falls in the first category as the use of CF by *B* violates the assumptions related to the use of OCS by *A*. Our

environment allows for the implementation of different strategies for conflict detection and resolution between features, and also more generally between agents.

3.3 Examples of CLIPS Rules and MediaPath/Micmac Outputs

An agent head can be described as one or many CLIPS engines that contain local facts and rules. In our example, we have one type of agent (User Agent) with two possible roles (Call-Side and Answer-Side), and four types of engines. Two of the latter, namely OCS (Figure 4a) and CF (Figure 4b), are briefly discussed here.

a) Originating Call Screening

```
(defrule prohibit
          (declare (salience 2))
          ?c <- (callTo ?to)
          (prohibit ?to)
=>
          (doProhibit)
          (retract ?c))

(defrule permit
          (declare (salience 1))
          ?c <- (callFrom ?from)
=>
          (doPermit)
          (retract ?c))
```

b) Call Forwarding

```
(defrule forward
          ?c <- (call)
          (forward ?to)
=>
          (doForward ?to)
          (retract ?c))
```

Figure 4 CLIPS rules for OCS and CF.

OCS has two rules with different local saliences. A rule is fired when facts match the LHS of the arrow (`==>`), and then actions on the RHS are performed. Required facts (or pre-conditions) are asserted when the agent's body peeks at the tuple-space. Rule `prohibit`, which has priority over rule `permit`, is invoked when a call connection is requested and when the destination (`?To`) is on the screening list. Relating this to our example, *A*'s screening list (within his instance of this OCS CLIPS engine) would be a fact-list containing (`prohibit` *X*). Upon firing, the rule will first send a comment to the tuple-space by calling a `doProhibit` function handled by the Java body. Then, it will retract all facts matching `callTo` *X* in its fact-list.

The `permit` rule of OCS only requires a call connection request in order to be invoked. Upon firing, the Java function `doPermit` will sent a message (containing a PERMIT status) to the tuple-space, and then the matching facts get removed from the fact-list.

CF is composed of only one rule, which is very similar to OCS' `prohibit` rule. Rule `forward` is fired upon the arrival of a call connection request and the presence of one specific fact that asserts the destination (bound to `?To`) to which calls should be forwarded. The RHS also makes use of an action that sends a comment to the tuple space (`FORWARDTO To`), followed again by a removal of all matching facts from the fact-list.

The ORIGINATING and TERMINATING features also use similar CLIPS rules. TERMINATING has two of them: `busy` will emit a REJECT comment when the Answer-Side is busy, while `not-busy` will simply PERMIT the call. ORIGINATING will reason about the comments related to its intention by cancelling (prohibiting or rejecting), forwarding, or establishing (permitting) the call. Four rules, with saliences corresponding to the different types of input comments, implement this strategy.

The next two figures illustrate executions of these rules in the prototyping environment.

On the left, MediaPath provides telephone keypads that interface with the user for number dialling and for ringing. Three stations are used for our example: 2000 is the call originator (*A*), 2001 the forwarder (*B*), and 2002 the answerer (*X*). These logical phone numbers are mapped to physical addresses 82000, 82001, 82002, which can be connected through MediaPath either to software phones or to real physical devices.

Figure 5 presents a simple case where *A* attempts to call *X* directly, and OCS prohibits the call request. The user agent 2000 assumes the Call-Side role, and two windows are shown, one per CLIPS engine in the head (other may also exist). Stations 2001 and 2002 are user agents that both assume the Answer-Side role. Their heads also contain concurrent CLIPS engines, some of which are in the figure. New tuples in the blackboard, not shown here for simplicity, cause facts to be asserted in local CLIPS engines by Java bodies. All facts are ordered according to their priority, as explained in Section 3.2. After a 2000-to-2002 connection proposal sent by ORIGINATING 2000, TERMINATING 2002 fires the `not-busy` rule and suggests permitting the call, while OCS 2000 fires its `prohibit` rule and suggests prohibiting the call. OCS 2000 does so because 2002 is on its screening list (represented by a fact in its local list). As a result, ORIGINATING 2000 decides to prohibit the call (rule `prohibit`) as this option has the highest local salience (4). All engines then retract the necessary facts from their respective fact-lists, and finally reinitialize themselves (not shown in the figure). The user agent 2001 was not involved in this conversation.

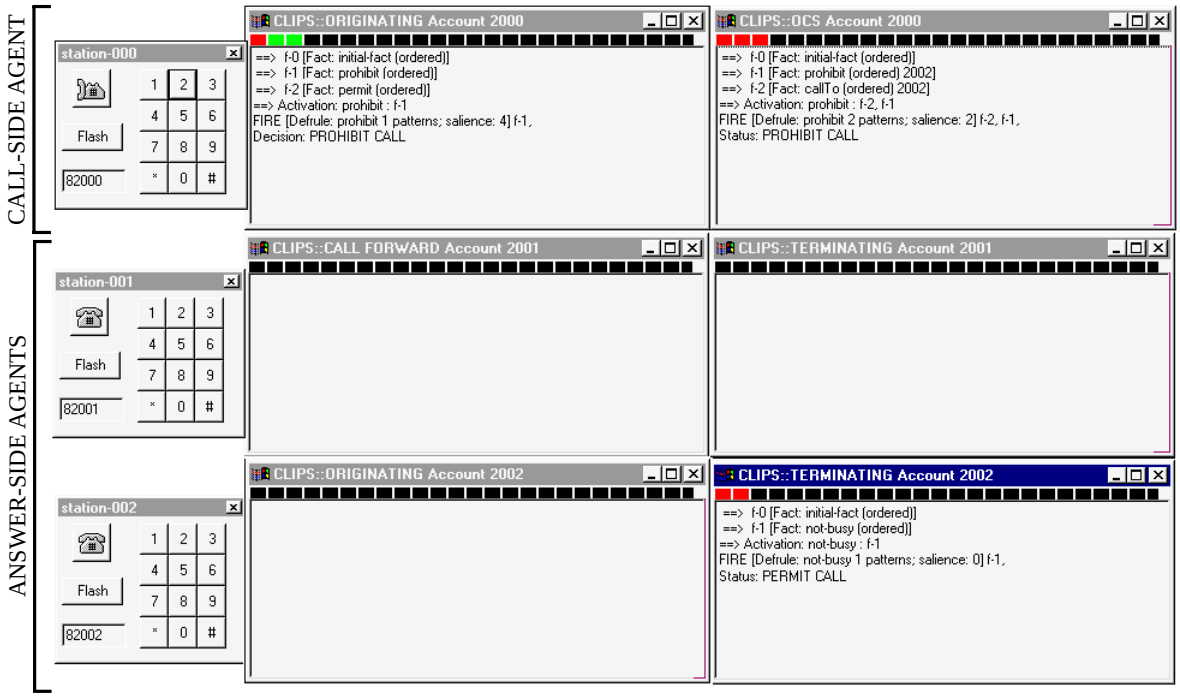

Figure 5 Micmac screen showing a call prohibited by OCS.

In Figure 6, *A* attempts to call *B*, whose CF feature forwards all calls to *X*. Upon the 2000-to-2001 connection proposal sent by ORIGINATING 2000, three comments (annotated by *(C1)*) are concurrently generated by other features. OCS 2000 posts a PERMIT (as *B* is not on *A*'s screening list), TERMINATING 2001 does the same, and CF 2001 posts a FORWARDTO (to *X*). Then, ORIGINATING 2000 takes the decision to forward this call *(D1)* and therefore proposes a 2000-to-2002 connection, as already illustrated in Figure 3. Although TERMINATING 2002 permits this call, OCS 2000 prohibits it *(C2)*. Therefore, since the PROHIBIT comment has the highest salience in ORIGINATING, the call is cancelled, as it should be *(D2)*. This last part of the scenario is the same as the one in Figure 5.

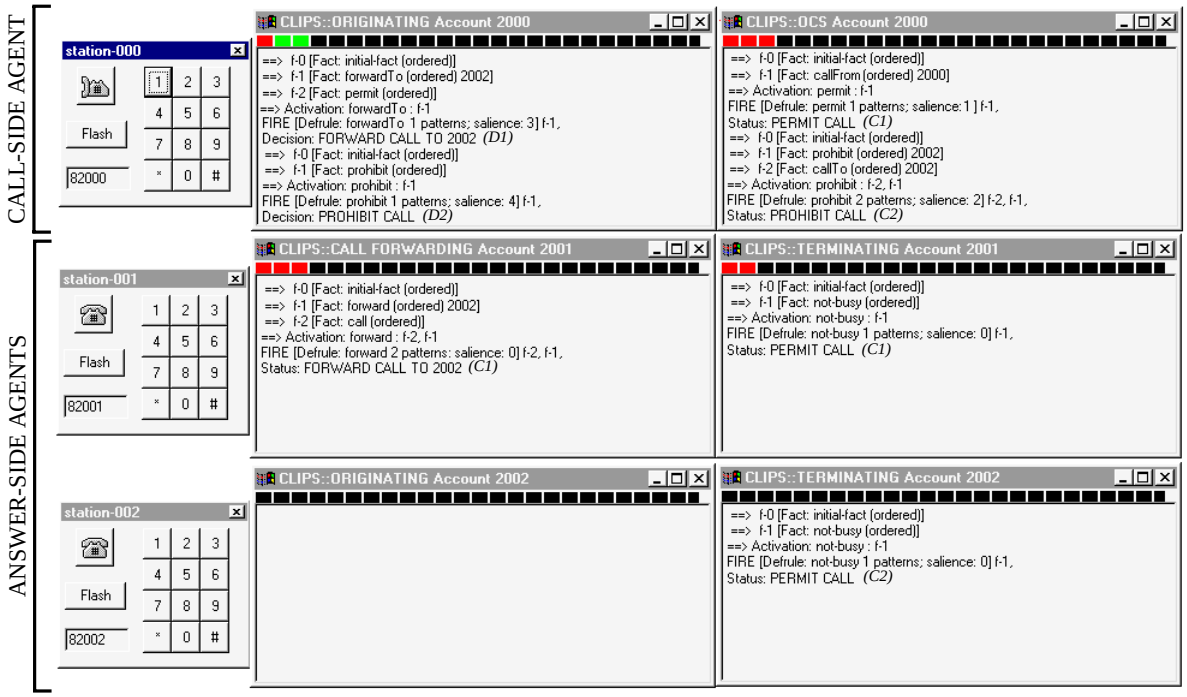

Figure 6 Micmac screen showing a call prohibited by OCS through CF.

4. Prototypes from High Level Agent Models

The most significant property of our approach is that systems are developed through a series of levels of abstraction in which humans, with machine assistance, can manipulate abstractions at one level into the ones at the next lower level. The full functional behaviour of an agent system is provided through high level models that express aspects of agents such as goals, beliefs, plans, jurisdictional relationships, usage rights, contracts, and conversations. In this section we present the high level agent models that allowed us to build the prototype described in the previous section.

Four agent models are used in our approach. Two of them are used to create the prototype and are fully described in this section, namely the agent internal model and the conversational model. The *agent internal model* defines the agents in terms of their internal structure and behaviour. The *conversational model* presents the coordinations among the agents. The other two, not described in this paper, are: the relationship model and the contract model. The *relationship* model describes inter-agent dependencies and jurisdictional relationships. The *contract model* defines a structure that translates one agent requirements to another. Contracts can be created when agents are instantiated or later on as needed.

4.1 Agent Internal Model

The agent internal model describes the mental state of agents. The agents are described in terms of their *goals*, *beliefs*, and *tasks*. We use tables to describe each agent's internal structure. Table 1 shows the agent internal model used to create the logic for the prototype. The goal column lists the goals an agent may adopt to reach a desired state. The precondition column lists the beliefs that should hold in order for the goal to be executed. The postcondition column lists the effects of executing a successful goal on agent's beliefs. The task column lists all the agent tasks, including subgoals, that are required to fulfill each goal. The latter may be decomposed

into subgoals which provide detailed or alternate ways of achieving that goal. These subgoals are shown in the tasks column as well as in the goal column. Each row in the agent internal model table represents a plan that can be instantiated at runtime to fulfill a goal. A goal may have different plans that can fulfill it, and hence each goal may have more than one entry in the table. The comment column contains a textual explanation of each plan. The agent internal model is derived from UCM models which are to be described in Section 5.

Table 1: Agent internal model for our telephony example.

	Goal	Precondition	Postcondition	Task	Comment
1	Process originating call	Number is collected	Request sent to answerer	send_request	ORIGINATING
2	Process originating call	Outgoing call connection requested	Call permitted or rejected	check_list doPermit doReject	OCS
3	Process call request	There is an incoming call	Caller and/or answerer are notified	ring notify_caller	TERMINATING
4	Process call request	CF is on. There is an incoming call	Caller notified of a new destination	doForward	CF

There are two ways to implement this table. One way is to let the user agents decide, at run-time, what features to invoke based on the feature's preconditions. In this case, only the features selected by the agent are activated. The other way consists in having all features simultaneously active, where each one can respond to proposals by others. In this type of implementation, a user agent chooses its actions based on the reactions of the different features. In this prototype, we chose the second approach, because it allows the dynamic creation, modification, and removal of features with minimum effect on the agents and the overall system maintenance.

The mapping from the agent internal model to the code described in Section 3 is straightforward in this example. Each row (feature) is implemented as a competing CLIPS engine. The relationship between this model and the generated code can be described by examining lines 2 and 4, and Figure 4.

From line 2 in the agent internal model, the code in Figure 4a is generated. Asserting the '`(callTo X)`' fact is the same as declaring the outgoing call connection request precondition as true. Here checking the list is stated as fact matching in the `prohibit` rule. We permit a call by having a lower salience `permit` rule that only fires if there is no prohibiting fact that matches in the `prohibit` rule.

From line 4 in the agent internal model, the code in Figure 4b is generated. Asserting the '`(call)`' fact states that there is an incoming call, as per the precondition. Asserting a '`(forward ?To)`' fact states that Call Forwarding is on, and that calls should be forwarded to the destination (`?To`). The forward rule is invoked if the preconditions are true, forwards the request, and retracts the incoming call condition. This last action we interpret as asserting the postcondition that the caller has been notified of the new destination.

4.2 Interagent Conversational Model

The purpose of the conversational model is to identify what messages are exchanged in order for the agents to cooperate and negotiate with each other. The conversational messages exchanged are determined by the plans, captured in the agent internal model, that are instanti-

ated by agents. Table 2 shows the conversational model used to generate the messages exchanged between agents. The model is described in a tabular format. This table has three columns: *received*, *sent*, and *comment*. The received column lists the messages received by the agent. The sent column lists all possible responses to each received message. The comment column contains a textual explanation of origins of messages.

Table 2: Interagent conversational model for our telephony example.

	Received	Sent	Comment
1		Prop(:connectFrom a :connectTo b)	ORIGINATING
2	Prop(:connectFrom a :connectTo b)	ACCEPT | REJECT	OCS
3	Prop(:connectFrom a :connectTo b)	ACCEPT | REJECT	TERMINATING
4	Prop(:connectFrom a :connectTo b)	CProp(:connectFrom a :connectTo f)	CF
5	CProp(:connectFrom a :connectTo f)	Prop(:connectFrom a :connectTo f)	ORIGINATING

The conversational model shown in Table 2 includes four types of messages, namely: Prop, CProp, ACCEPT, and REJECT. The set of the four messages implements a generic agent negotiation mechanism. In our system, an agent that wants to communicate with another agent sends a proposal (Prop message) and waits for a response. The responses the agent can get to its proposal can be a counter proposal (CProp), proposal acceptance (ACCEPT), or proposal rejection (REJECT). If an agent gets a proposal or a counter proposal, then it needs to evaluate the proposal and send a response back.

Line 4 of the conversational model shows that when an agent (an answerer), who is subscribed to Call Forwarding, receives a connection proposal, it responds by sending a counter proposal recommending to connect to user agent *f* instead. User agent *f* is the designated agent for handling the calls of the receiver of the original proposal. Line 5 shows the response of the user agent to the counter proposal in line 4. The user agent accepts the counter proposal by initiating a new connection proposal to *f*. Note that *a*, *b*, and *f* are formal parameters in this table.

The conversational messages captured by the model are implemented as shown in Figure 3. Proposals correspond to tuples, and responses to proposal are implemented as tuples with status fields. The ACCEPT message is implemented by the PERMIT status and the REJECT message by the PROHIBIT status in tuples. The counter proposal in line 4 is implemented by the Tuple(:FORWARDTO f) status. Note that in the implementation, a new field (callID) is added to each message to distinguish the different call sessions.

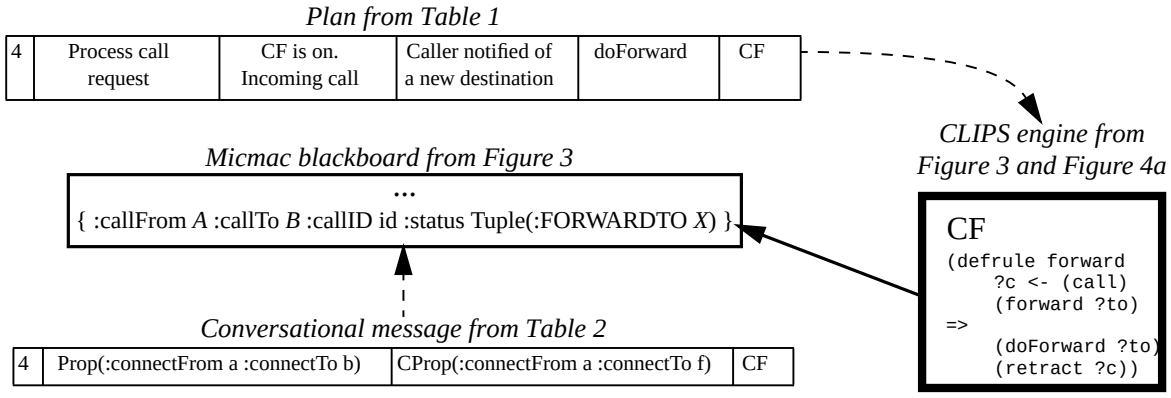

Figure 7 From internal and conversational models to CLIPS rules and messages.

Figure 7 summarizes the transformation of plans from the agent internal model into rules of an independent CLIPS engine. In this specific figure, the Call Forwarding engine is found

within the head of the user agent (Answer-Side role). It also illustrates a message sent by the CF feature to the Micmac blackboard at run-time. This message is formatted according the corresponding row in the interagent conversational model.

5. High Level Agent Models from a Scenario-Path Notation

5.1 Scenario Paths for Agent Systems

Scenario paths provide a means of representing the "structure of behaviour" for a whole system directly, in understandable diagrams that are above the level of messages and protocols. The diagrams (called UCMs) show wiggly lines depicting scenario paths superimposed on sets of boxes representing system components (e.g., agents). UCM paths start at points where events occur. They end at points where the effects of the events have ceased *actively* rippling through the system. In between, they touch components that perform responsibilities, in the causal order in which the responsibilities are performed. Composite diagrams that contain many possible paths, with common parts superimposed, provide a condensed view of many possible different behaviours, including ones involving concurrency.

UCMs do not *specify* behaviour, they describe its *path structure* in a way that enables a person to visualize scenarios by mentally moving tokens along the paths. Thus, reading UCMs is like playing a board game with multiple players and a token for each player, in which board layout and the game rules determine possible play scenarios. The UCM "player" determines the actual scenarios by following the paths and some rules (the rules are provided by the meanings of the notations, the nature of the system and its components, and map documentation).

Particularly useful for agent systems is that fact that UCMs are able to provide understandable descriptions of dynamic situations at the whole system level. This is accomplished by expressing UCMs as compositions of sub-UCMs that may be dynamically selected while the system is running. A *stub* notation indicates where sub-UCMs may be plugged in (the sub-UCMs are accordingly called *plug-ins*). A UCM with dynamic stubs shows a dynamically modifiable "behaviour structure". The structure is modified dynamically by selecting an appropriate plug-in for a stub when a scenario token reaches the stub (according to assumed system conditions at that point). For application to agent systems, these concepts may be easily related to BDI concepts (Section 5.3): stubs are related to *desires*; selected plug-ins are related to *intentions*; sets of plug-ins are related to *plans*; and conditions that select a plug-in are related to *beliefs*.

Figure 8 shows a UCM for a system of user agents that handles telephone calls for users in some network. The plug-ins represent telephony features, including default features for the ORIGINATING and TERMINATING ends of a call and also the two features illustrated in Figure 1, namely OCS (Originating Call Screening) and CF (Call Forwarding). The defaults are viewed as features at the same level as OCS and CF. Feature interaction results when certain combinations of plug-ins are selected under certain system conditions. The UCM allows a person to visualize the dynamic situations that give rise to feature interactions. The visualization is in terms that can be directly related to how agents are implemented.

Figure 8 shows, at its center, a path taken by a call request through a set of software agents to a phone ringing at some remote user location. The CSP (Call-Side Processing) and ASP (Answer-Side Processing) stubs have dynamically selected plug-ins for different features (both stubs are shown in both agents to show that the situation is symmetrical, in principle, although only one direction is shown). At the top of the figure is shown a set of plug-ins (the default ones) that cause no problems. At the bottom of the figure is shown a set of plug-ins that, when selected in this particular combination, may cause a feature interaction. The way to read

this diagram is to trace a normal call through the top set of plug-ins and to trace a forwarded call through the bottom set. One of the plug-ins is the same in both sets (the default TERMINATING plug-in), but is duplicated so the diagram can be read as suggested.

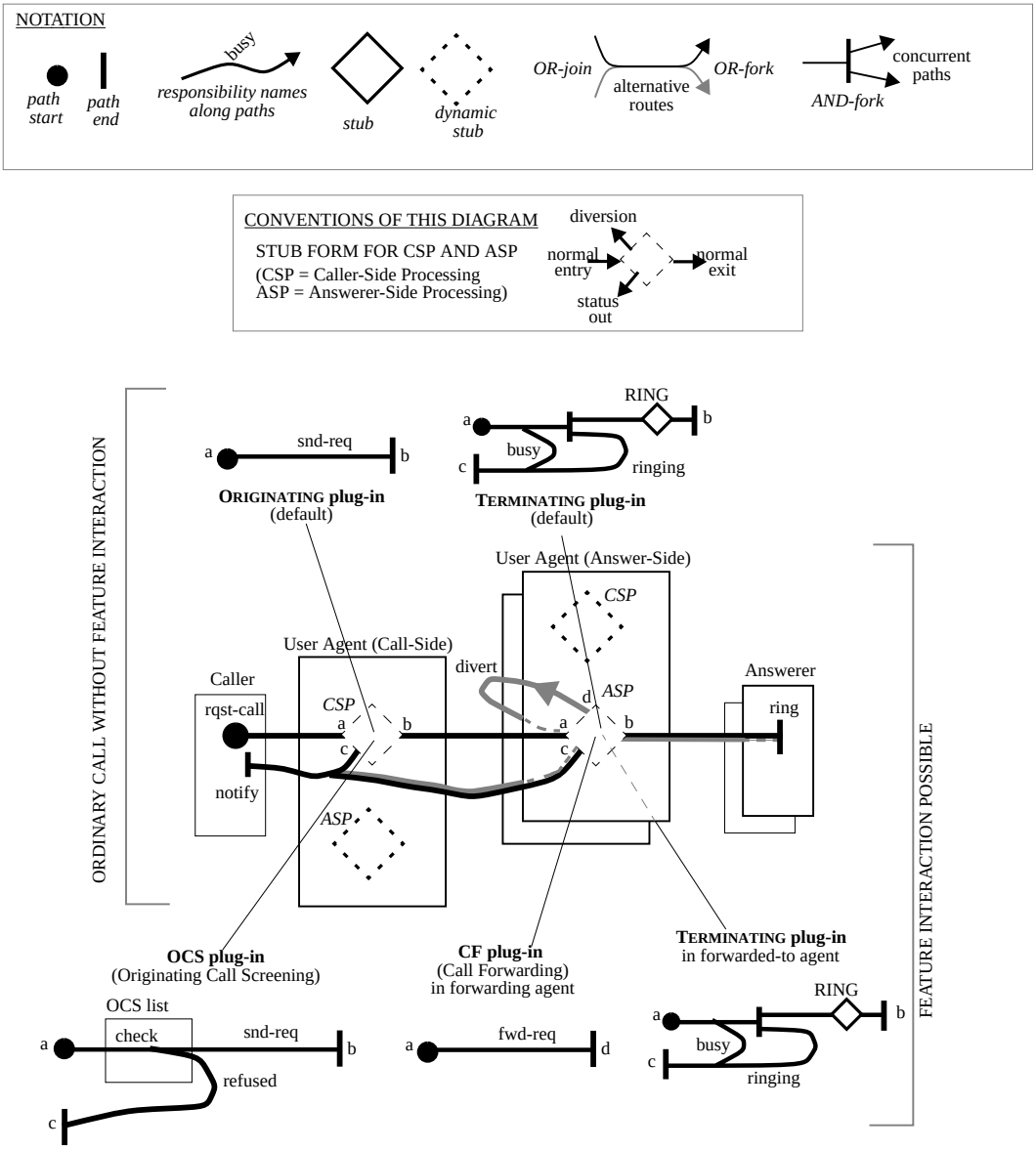

Figure 8 A telephony feature-interaction example.

The main UCM at the center of this diagram includes a path that winds through a set of Answer-Side user agents (the grey path that diverts from point *d* on the ASP stub). The grey shading underneath some of the black path segments emphasizes that this diversion causes the route to continue “underneath” for a different agent. The extra shading is not really necessary once you understand the meaning of the diagram (the diversion from *d* would not go back into the same agent, by definition, so what follows would have to involve a different agent). The map includes the possibility that several diversions within the Answer-Side set of agents may occur.

The default ORIGINATING plug-in describes the default behavior when the caller is not subscribed to any feature. The plug-in performs the *snd-req* responsibility which causes the caller to send a request for a call connection to the answerer user agent (that a request must be

sent is *inferred* from the fact that the next point along the path is the ASP stub in the answerer agent).

The default TERMINATING plug-in describes the default behavior when the answerer is not subscribed to any feature. The plug-in starts with an OR-fork. If the user is busy, the path labeled *busy* is followed and the caller is notified that the answerer is busy. Otherwise the scenario proceeds to an AND-fork. One path of this leads to a RING stub (for which no plug-in is provided here) that notifies the answerer, for example by ringing a phone device. The other path notifies the caller of call progress.

The OCS plug-in would be selected for the CSP stub when the caller is subscribed to the Originating Call Screening feature. The path begins by checking the OCS list. If the dialled number is on the list, then the connection is refused. This is shown on by the OR-fork that follows the *check* responsibility. Otherwise the caller is allowed to connect to the dialled number.

The CF plug-in would be selected for the ASP stub when the answerer is subscribed to the Call Forwarding feature, and system conditions at the time of entry to this stub select this feature. The CF feature performs the *fwd-req* responsibility which causes the incoming call to be forwarded to another user agent, which will be responsible for processing the original call request.

A UCM such as Figure 8 expresses the notion that the different features (represented by plug-ins) are competitors to fulfill the functional behaviour required by stubs. Such a UCM makes no commitment to how the competition is to be resolved. It could be resolved by selecting only one feature. However, the approach of Section 3 implements the different features as concurrent, competing rule engines that resolve the competition dynamically. Observe that UCMs make no commitment to which approach is taken.

5.2 Scenarios With and Without Feature Interaction

Specific scenarios may be expressed in path terms by selecting a path of interest, selecting a particular set of plug-ins for the stubs of the path, and redrawing the path to include the plug-ins explicitly. Figure 9 and Figure 10 show two trouble-free UCMs that result from doing this for Figure 8.

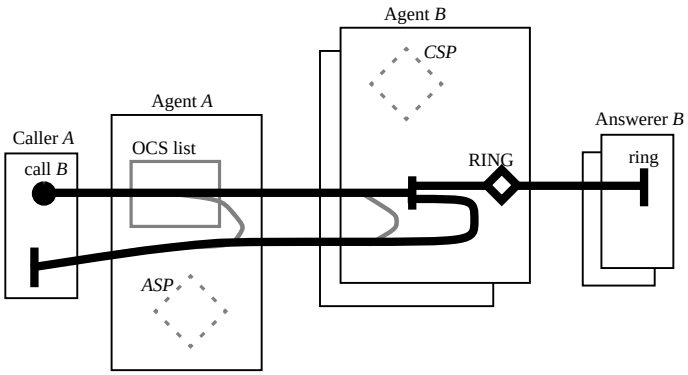

Figure 9 Path view of "*A* connects to *B*" scenario.

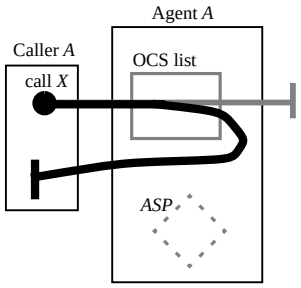

Figure 10 Path view of "*A* refuses call to *X*" scenario.

Figure 11 shows a feature-interaction-prone UCM involving Originating Call Screening and Call Forwarding. This is a feature interaction if *X* is on *B*'s OCS list.

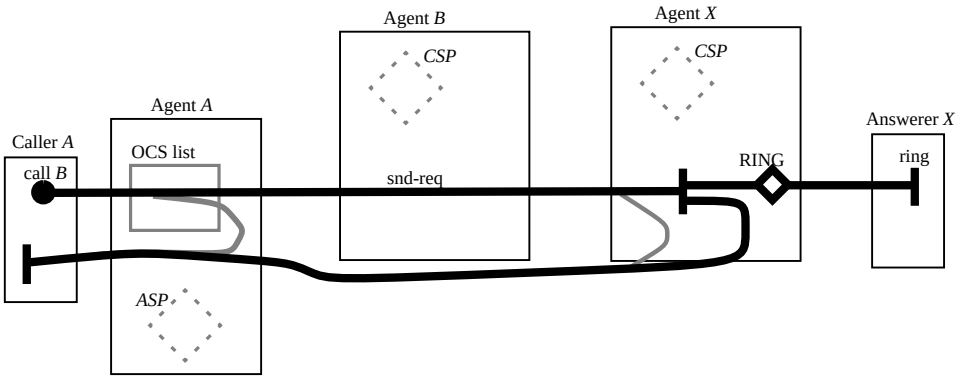

Figure 11 Path view of "*A* allowed to call *X* scenario"
(a feature interation if *X* is on *A*'s OCS list).

Figure 12 shows a different stubbed UCM (compare with Figure 8) that avoids any possibility of the above feature interaction by routing the forwarding path back through the calling agent to check if the intended forward-to number is forbidden (in general, redesigning a main UCM like this could require redesigning the plug-ins). This is the UCM that was implemented in our prototype, not the one in Figure 8. We can observe that the paths in Figure 12 emerge from the rules and the execution scenarios in the simulation (Figure 4 and Figure 6).

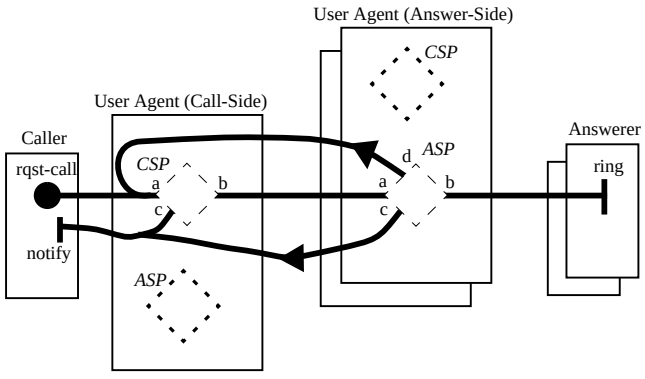

Figure 12 Feature interaction resolution.

Thus UCMs help a person to visualize, reason about, and resolve feature interaction problems in systems of agents at a high level of abstraction. The close relationship of UCMs to BDI models of agents enables this thinking to be related to agent implementations in a systematic manner, as will now be explained.

5.3 From Use Case Maps to Intermediate Agent Models

The nature of the relationship between UCMs and the intermediate agent models of Section 4 is summarized in Figure 13. Path segments that traverse an agent represent goals, dynamic stubs along the path represent sub-goals, static stubs represent sets of agent tasks, path preconditions and postconditions help in forming the belief set, and responsibilities along the path constitute the agent's high level tasks. The causal sequences in UCMs continue to be causal sequences as far as the agent internal model is concerned (e.g., the concept is that a task in one agent may *cause* tasks of other agents to be activated—the causal mechanism is defined by the conversational model).

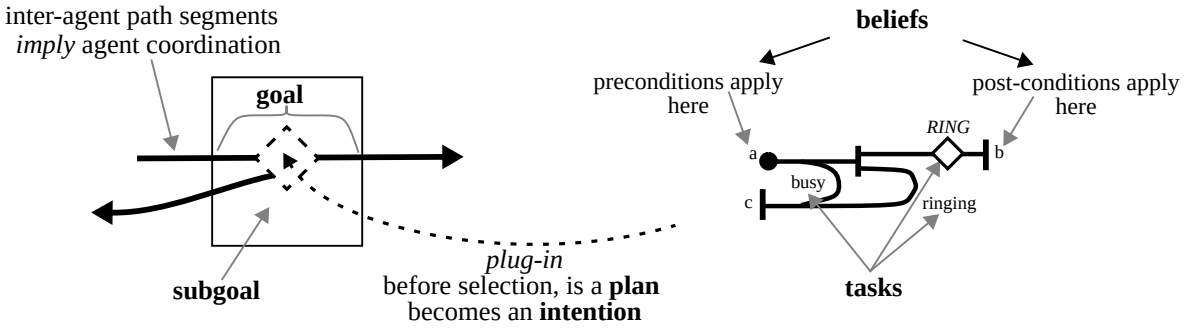

Figure 13 From UCMS to agent internal and conversational models.

UCMs are, in general, incomplete as system specifications, so human intelligence is required to produce agent internal models from UCMs. The closeness of the concepts is helpful, but the process is not simply one of linearly filling in details. For example, if the starting point is Figure 8, paths crossing a user agent in both directions must be mentally combined to get an agent-centric picture that covers all the possibilities expressed by the UCM. For simplicity, we shall deal with only one direction in the following explanation.

The UCM of Figure 8 leads to the agent internal model of Table 1, as follows. The high level basic call UCM shows that each user agent has two dynamic stubs: CSP and ASP. Each of these stubs is mapped into a goal in the agent internal model. Since there are two plug-ins that can satisfy each stub, two rows (plans) for each goal are needed in the agent internal model to represent each plug-in (for a total of four rows). The preconditions, postconditions, and responsibilities for each plug-in are captured in the corresponding row (plan). Because, in this example, the only UCM entity along the path segments traversing the agent is a single dynamic stub, only the stub is represented as a goal.

The *need* for coordination is implied in UCMs by inter-agent path segments. The actual coordination *mechanism* is described by the conversational model, which identifies the messages to be exchanged for this purpose. Protocols types are identified for different purposes. The plans captured in the agent internal model (Table 1), in conjunction with the protocol identification, were used to construct the conversational messages in Table 2.

6. Discussion

6.1 Putting the Pieces Together

The approach has been described from the bottom up. Here is how we see it from the top down:

1) UCMs are used to discover agents and their high level behaviour. They give the system

picture in a way that includes dynamic situations explicitly. UCMs are precise structural entities that contain enough information in highly condensed form to enable a person to visualize system behaviour.

2) A relatively conventional agent internal model is derived partly from UCMs, partly from human input, and partly from standard patterns.

3) The conversational model is derived from the coordination in UCM models and from the agent internal model. These intermediate models aim to provide the transition between UCMs and implementations.

4) Each plan from the agent internal model is transformed into rules of an independent CLIPS engine, following the message syntax suggested in the conversational model. This leads to executable high level prototypes that include only some aspects of practical agent systems (the aspects concerned with controlling dynamic situations involving multiple agents).

A novel aspect of our approach is its constructive nature. Systems are developed through a series of levels of abstraction in which humans, with machine assistance, can manipulate abstractions at one level into abstractions at the next lower level.

We are using this approach to investigate some difficult issues in the design and development of practical agent systems. We hope that this approach will help customers and system designers to communicate better about requirements, provide a systematic process for transforming requirements into metalevel agent logic (and hence into software implementations), and help with system evolution by providing a high level reference for making detailed changes.

We have used relatively familiar BDI concepts here but are also investigating the OPI (Obligations-Permissions-Interdictions) approach of Barbuceanu [2] (which also appears in these proceedings). BDI models are agent-centric, meaning they only express agent internal beliefs, desires and intentions. They have little to say about collective behaviour, leaving it to be whatever emerges from the actions of unconstrained individuals. The OPI approach, on the other hand, includes constraints on collective behaviour. The approach is based on deontic logic and deals with the propagation of jurisdictional assertions and the resolution of design and run time conflicts in agencies. It seems to us that the presence of such information in our intermediate models may improve the understandability and explainability of agent systems. The combination of visualization with UCMs and formal logic with OPI models could be very powerful in this direction.

Feature interaction is not our only interest, but its difficulty makes it very suitable for getting a handle on the issues. Rule-based prototyping and blackboards are convenient, available technologies for trying out the ideas. There is nothing in the UCM-based approach that *requires* them. In particular, the blackboard approach is not fundamental. The blackboard only provides a simple means of simulating all forms of communication needed in our models, without becoming bogged down in actual communication software and hardware. The blackboard allows us to simulate, in one conveniently uniform way, communication between agents, communication between agent heads and agent bodies, and communication between rule engines in agent heads.

6.2 Scalability Issues

A key issue is scaleup. The running example of this paper illustrated only a small number of features. While this was enough to convey the concepts, more features must be included to

demonstrate practicality. We are currently experimenting with adding more features to our models. So far, the results are encouraging. The UCM modelling effort does not yet seem to blow up as the number of features increases, because similarities in path signatures start to emerge at both the stub and plug-in level (such signatures are a form of UCM "pattern"). We are hopeful that such patterns will reduce the combinatorial problem to manageable proportions in UCM models.

Our agent-based approach also allows for the division of the system's hundreds of features into agents that contain far fewer. This division implies feature interaction avoidance since it greatly limits the number of features which have to be considered for resolution both at run and design time. Additionally, the agent framework provides patterns tuned to solve the specific problems which make up the feature interaction problem. For example, resource allocation problems are managed by several patterns which are both inside of and between agents.

The BDI and OPI models lead to scalability in the sense of enabling us to build systems, local parts of which can be customized, personalized and even evolved without reference to other parts. However, we must be sure that the result will still perform well. It may appear that the blackboard approach used in our prototyping environment cannot do this. We explained earlier that we only used blackboards as a simple way of modeling all kinds of communication, without necessarily committing to them for actual applications. However, this does not mean blackboards cannot be practical in networks. High speed communications and KQML are changing assumptions about the lack of scalability of blackboards. With these developments, making blackboards practical should be possible by techniques such as structuring them in a hierarchical way and dynamically creating and destroyed localized ones when required.

7. Conclusions

Although telecom systems are a fertile field for agents, the dynamic nature of agent solutions presents great difficulties for telecom software designers. This paper makes the following contributions to helping telecom software designers bridge a large conceptual gap between dynamic situations involving multiple agents and software design details of individual agents. Use Case Maps (UCMs) are offered as a way of making complicated dynamic situations involving multiple agents not only visible and understandable at a glance, but also directly translatable into BDI-style solutions. A systematic way is offered of translating UCMs into such solutions and thence into high level, executable system prototypes. A prototyping environment is described. It enables issues such as dynamic conflict resolution to be explored.

The practicality of the approach is illustrated by a practical problem: telephony feature interaction. The problem is an appropriate one for the following reasons: feature interaction is a conflict resolution problem of a type suitable for agents; the telephony feature interaction community has recognized agents as a suitable solution technique; and the problem has all the dynamic characteristics that make agent solutions difficult for telecom software designers. Contributions are made to helping telecom software engineers solve this problem with agents. We show how to represent features in UCM terms as plug-ins for dynamic stubs. We describe how the UCM view of competing feature plug-ins may be implemented as competing CLIPS rule engines communicating via a blackboard. We offer this as a scalable approach that permits adding new features incrementally at run-time; this is important for the type of telecom applications that interest us, where no deterministic or optimal solution is known, and where opportunistic problem solving seems to be the most practical approach.

This paper demonstrates the essence of an intended approach for future commercial systems.

Acknowledgments

This research is supported by Mitel, TRIO (now CITO), and NSERC. Contributions are gratefully acknowledged of Andrew Miga, Gabriela Alexiu, Debbie Pinard, Peter Perry, and Michael Weiss.

References

[1] M. Barbuceanu, M.S. Fox, *COOL: A Language for Describing Coordination in Multi-Agent Systems*, in Proceedings of the International Conference on Multi-Agent Systems, San Francisco, CA, 1995

[2] M. Barbuceanu, T. Gray, S. Mankovski, *How To Make Your Agents Fulfil Their Obligations*, Third International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents (PAAM'98), 23-27 March 1998, London, UK

[3] J. Bergstra, W. Bouma, *Models for Feature Descriptions and Interactions*, Feature Interactions in Telecommunication Networks IV, IOS press, pp. 31-45, 1997.

[4] R. J. A. Buhr, R. S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996.

[5] R.J.A. Buhr, *Use Case Maps for Attributing Behaviour to Architecture*, Proc. Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), April 15-16, 1996, Honolulu, Hawaii.
http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/attributing.ps

[6] R.J.A. Buhr, A. Hubbard, *Use Case Maps for Engineering Real Time and Distributed Computer Systems: A Case Study of an ACE-Framework Application*, Hawaii International Conference on System Sciences (HICSS'97), January 7-10, 1997, Wailea, Hawaii. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hicss.ps

[7] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, D. Pinard, *Understanding and Defining the Behaviour of Systems of Agents, with Use Case Maps*, Carleton report, poster session, Second International Conference and Exhibition on Practical Applications of Intelligent Agents and Multi-Agents (PAAM'97), London, April 1997.
http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam97.ps

[8] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, *A High Level Visual Notation for Understanding and Designing Collaborative, Adaptive Behaviour in Multi-agent Systems*, Hawaii International Conference on System Sciences (HICSS'98), Hawaii, January 1998. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/agents-ucms.ps

[9] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, *Applying Use Case Maps to Multi-agent Systems: A Feature Interaction Example*, Hawaii International Conference on System Sciences (HICSS'98), Hawaii, January 98.
http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hiccs98.ps

[10] R.J.A. Buhr, *High Level Design and Prototyping of Agent Systems*, Research project description. http://www.sce.carleton.ca/rads/agents/

[11] F.M.T. Brazier, C.M. Jonker, J. Treur, *Principles of Compositional Multi-Agent System Development*, report, Vrije Universiteit Amsterdam, Department of Mathematics and Computer Science, 1997.

[12] E.J. Cameron et al., *A Feature Interaction Benchmark for IN and Beyond*, in Feature Interactions in Telecommunications Systems, IOS press, pp. 1-23, 1994.

[13] Ernest Friedman-Hill, *Jess: The Java Expert System Shell, Version 3.1*, Sandia National Laboratories, 1997. http://herzberg.ca.sandia.gov/jess/

[14] Joseph C. Giarratano, *CLIPS User's Guide, Version 6.05*, Software Technology Branch, NASA, JSC-25013, November 1997. http://www.jsc.nasa.gov/~clips/CLIPS.html

[15] Guideware Corporation, *The Implementation of Business Processes with Mobile Agents*, Technical Paper, Guideware Corporation, Mountain View, California, 1995.

[16] E. Kendall, *A Methodology for Developing Agent Based Systems for Enterprise Integration*, IFIP Working Conference of TC5 Special Interest Group on Architectures for Enterprise Integration, Queensland, Australia, November 1995.

[17] Mitel Corporation, *MediaPath*, 1997. http://www.mitel.com/MediaPath

[18] Mitel Corporation, *Micmac*, 1997. http://micmac.mitel.com/

[19] D. Pinard, M. Weiss, T. Gray, *Issues In Using an Agent Framework For Converged Voice/Data Applications*, Second International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents (PAAM'97), 21-23 April 1997, London, UK

[20] A. Rao, M. Georgeff, *BDI Agents from Theory to Practice*, Technical Note 56, AAII, April 1995.

[21] Y. Shoham, *Agent-Oriented Programming*, Artificial Intelligence, 60(1), pp 51-92, 1993.

[22] M. Weiss, T. Gray, A. Diaz, *A Middleware for Developing Distributed Multimedia Applications*, Proceedings of the ISCA International Conference on Parallel and Distributed Computing, Orlando, Florida, 1995.

[23] M. Weiss, T. Gray, A. Diaz, *An Agent Based Distributed Multimedia Service Environment*, International Conference on Tools with AI, Herndon, Virginia, 1995.

[24] M. Weiss, T. Gray, A. Diaz, *A Service Environment for Distributed Multimedia Applications*, Workshop on Distributed Information Networks, IJCAI 95, Montréal, Canada, 1995.

[25] M. Weiss, T. Gray, A. Diaz, *Experiences with a Service Environment for Distributed Multimedia Applications*, Feature Interactions in Telecommunication Networks IV, IOS press, pp. 242-256, 1997.