# Feature-Interaction Visualization and Resolution in an Agent Environment

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**. Our project is concerned with how dynamic agencies (sets of collaborating agents that vary their composition and collective behaviour over time) may be used to resolve demanding telecom problems in a flexible manner. Feature interaction (FI) is being studied as an instance of such problems for which dynamic agencies may be an appropriate solution. In our approach, Use Case Maps (UCMs) provide system-wide "behaviour structures" that enable people to get a global understanding of dynamic situations. Feature interactions can be seen visually in the UCMs and then be reasoned about and resolved by people at the UCM level. Tables generated from these behaviour structures provide a framework for humans to add information that will enable executable prototypes to be generated. These executable prototypes are FI-avoidant systems where features are modelled as competing rule engines and interactions are detected and resolved at run time by coordinating through a blackboard. The approach offers the promise of being scalable to practical numbers of features and is being considered for use in future commercial systems.

## 1. Introduction

Whether features are implemented with conventional software techniques or with techniques centering around agent metamodels of various kinds, dynamic situations of the kind that give rise to feature interactions are, in the software development process, relegated to details of software design diagrams and code. As a consequence, the overall behaviour is left to be whatever emerges at run time. Feature interactions may then come as a surprise. Ways are needed of making feature interaction problems more visible to software developers and of enabling software solutions to flow from high level descriptions.

In our work, the visibility problem is addressed with a technique called Use Case Maps (UCMs) [3][4][6][7]. UCMs provide system-wide path structures that enable people to get a global understanding of large scale dynamic situations. Feature interactions are seen visually in the UCMs and can be resolved by design at the UCM level. This approach arises from a view that feature interaction is an inherent problem of distributed software that needs resolution techniques accessible to software designers and developers. The approach rises above details, such as state transitions within components of the system and interactions between components, while still preserving the essence of the problem, which is the composition of events and sequences in different places in the system in unexpected ways.

Tables partially generated from the UCMs provide a framework for humans to add information that enables executable prototypes to be generated. In the executable prototypes, features are modelled as competing rule engines and interactions are resolved at run time by coordinating through a blackboard. The blackboard is a prototyping convenience, not necessarily a proposed practical implementation technique.

We are presenting a technique which can be used to *detect* and *resolve* feature interactions at design time (i.e., the UCMs and tables), empowered by an FI-*avoidant* agent architecture that provides patterns for detection and resolution at run time (this extends a theme presented in [18]).

For illustrative purposes, we use a running example with two common features, Originating Call Screening (OCS) and Call Forwarding (CF) [9]. OCS forbids calling numbers on a screening list, while CF forwards incoming calls to another number. A feature interaction occurs if some user *A*, whose OCS screening list includes another user *X*, calls user *B*, who forwards calls to *X* through CF. This is only an example; we have applied and are continuing to apply the approach to other features and combinations of features not described here.

Although our approach begins with UCMs as the top level view, for clarity of exposition to an audience not familiar with UCMs, the ideas are developed from the bottom up. Section 2 describes how feature interactions may be detected and resolved in an environment that combines Java agents and competing CLIPS (C Language Integrated Production System) rule engines that resolve feature conflicts through a Micmac blackboard (this environment includes a MediaPath system that simulates actual telephony). Section 3 describes how the rules and other elements of this environment are derived in a systematic way from tables. Section 4 describes how the tables flow from path descriptions in UCMs. Section 5 discusses where we are going with this approach and how we think others might use it. Section 6 draws conclusions.

A more extended treatment of the ideas in this paper is contained in [5]. This work is part of a project [8] that is concerned with the systematic design and implementation of dynamic agencies (i.e., sets of collaborating agents that vary their composition and behaviour over time). From the perspective of the feature interaction community, agents are just software components that contain logic to implement features and resolve feature interactions, and that communicate with each other in the terms required by the features.

## 2. A Prototyping Environment, By Example

### *2.1 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 [14], an open, standards-based communications server produced by Mitel. MediaPath is comprised of call control software and server telecommunication boards (voice processing, trunk, and line boards).

Micmac [15] 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 blackboard. Any other feature can then comment on the action and the originating feature can take this advice and decide how to proceed. The agent is reasoning about the development of an intention.

As shown in Figure 1, each agent is composed of a head and a body. The head is where decisions are taken (rules are selected), and the body is where these decisions are carried out

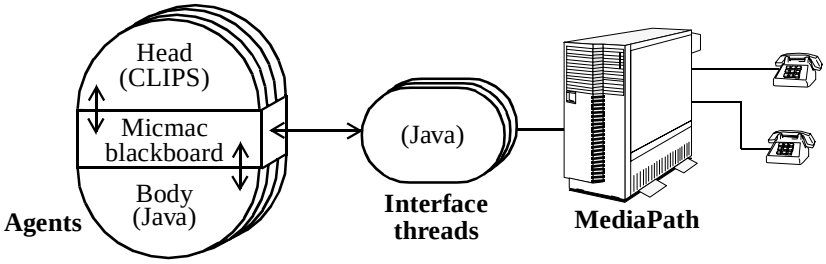


**Figure 1** Coordination between agents and MediaPath through a Micmac blackboard.

(tasks are executed). The head is implemented declaratively using the CLIPS expert system [12], and the body is implemented in Java. The head and the body of an agent also communicate through the blackboard. 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 (features) in agent heads. 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 [11], 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 that would describe a call request from *A* to *B* is `{ :connectFrom A :connectTo B :callID id }`. Identifier ingles such as `callID`, as well as other features of Micmac, allow for the dynamic creation of multiple logical blackboards inside one physical blackboard. They can also serve as a basis for implementing security mechanisms for blackboards.

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 the symbol '`?`', 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.

### *2.2 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 2 (ORIGINATING, OCS, TERMINATING, and CF), are instantiated as discrete entities which will interact through the posting of intentions on the tuple space. These intentions, shared between agents, can trigger advisors to prevent undesired feature interactions without explicitly programming a solution.

Agent heads implement these features as competing CLIPS engines. This approach based on a blackboard simplifies the interaction protocols for the collaboration among features instantiated within an agent. It also 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.

The undesirable interaction between OCS and CF, as stated in Section 1, is that *A* should not be allowed to connect to *X* through *B*. Figure 2 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 *A*'s default 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

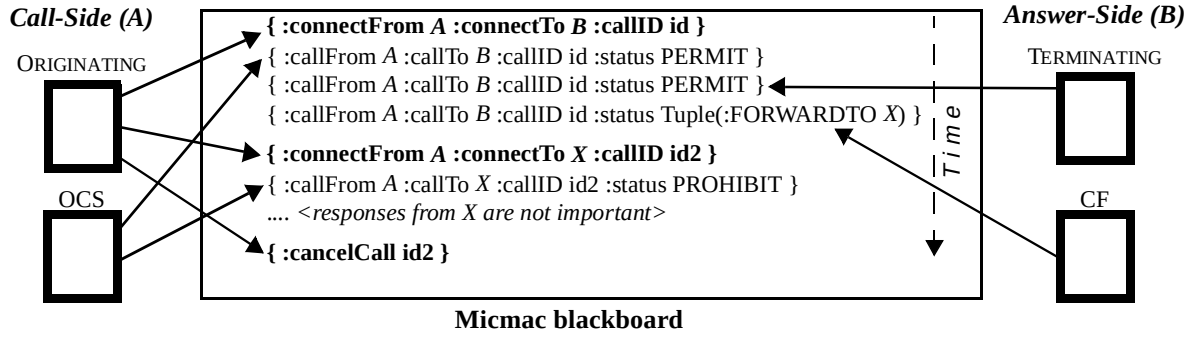


**Figure 2** Competing rule engines communicating via a blackboard.

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, 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. Comments that arrive late or that remain unused for a specific `callID` will be ignored and eventually removed automatically by the blackboard, after some predetermined period or some duration attached to these tuples.

This scenario illustrates the resolution of conflicts between features that are active simultaneously. Feature interactions can usually be of three kinds: violation of assumptions, indeterminacy, and data violation. The OCS/CF interaction 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.

The quick example developed here illustrates only a part of the mechanisms we want to implement in our framework. In future implementations, we plan on using distributed and dynamic blackboard that would enforce separation between agents and local decisions or concerns. In the current implementation, the ORIGINATING feature, located in the call-side agent, receives comments on a global blackboard from all the features in the answer-side agent. It is in charge to make the final decision while all other features act as advisors, whatever their location. This ORIGINATING feature should however receive only one comment from the answer-side agent, which would result from a local decision. Local decisions should be taken first in the call-side agent, and then in the answer-side agent (if necessary) before further negotiations are undertaken. However, for the termination of the negotiation

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 3** CLIPS rules for OCS and CF.

process to be guaranteed, one feature should always be in charge of making the final decision after some mutual agreement. All the other features would then go back to a safe state.

### *2.3 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 a single type of agent (User Agent) with two possible roles (Call-Side and Answer-Side), and four rule engines. Two of the latter, namely OCS (Figure 3a) and CF (Figure 3b), are briefly discussed here.

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 preconditions) 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 post (poke) a comment (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 similar in form 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 PROHIBIT 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 prohibiting, forwarding, or permitting the call. Three CLIPS 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 can be connected through MediaPath either to software phones or to real physical devices.

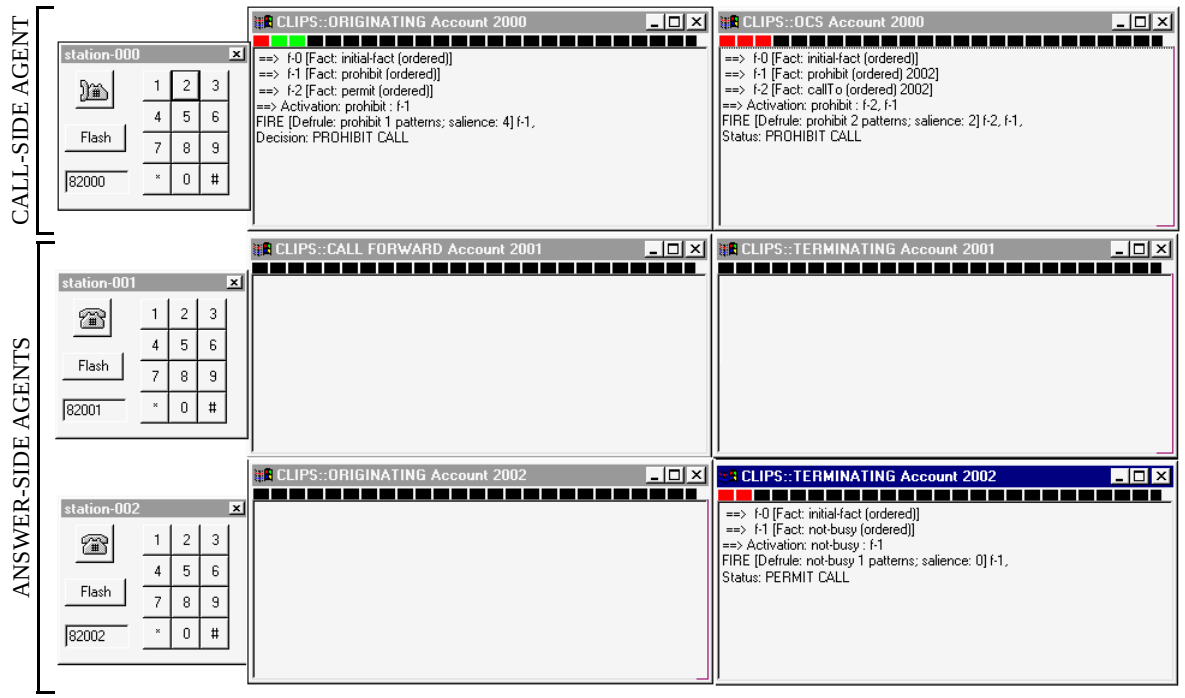


**Figure 4** Micmac screen showing a call prohibited by OCS.

Figure 4 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 2.2. 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

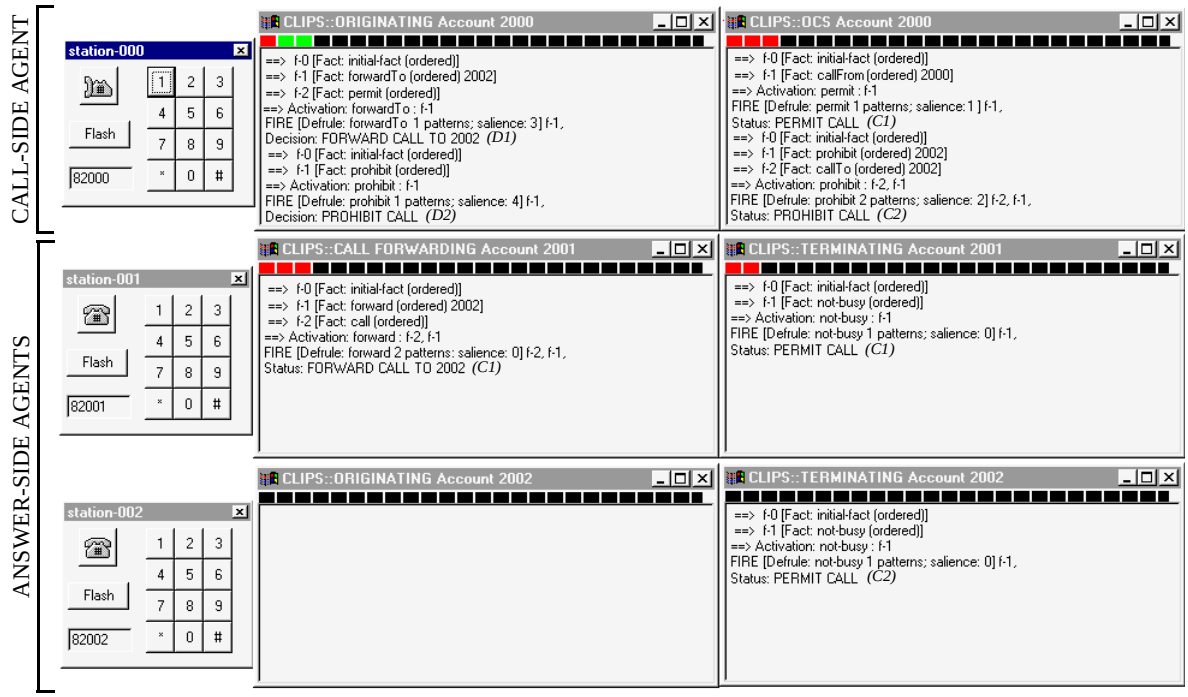


**Figure 5** Micmac screen showing a call prohibited by OCS through CF.

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.

In Figure 5, *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 previously illustrated in Figure 2. 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 4.

## 3. 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. Tables are used to express intermediate models between the prototypes of the previous section and the UCMs of the next section. Two intermediate models are needed for the running feature interaction example of this paper. The *agent internal model* defines the internal behaviour of an agent. The *conversational model* describes the coordination mechanism among agents. Other models not described here may also be needed.

### *3.1 Agent Internal Model*

Table 1 shows the agent internal model for the example of Section 2. In this particular table, each row represents a feature, identified by the comment column.

**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 prohibited | check_list<br>doPermit<br>doProhibit | OCS |
| 3 | Process call request | There is an incoming call | Caller and/or answerer are notified | ring<br>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 (not used in Section 2) 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 (used in Section 2) is to allow all features to be simultaneously active, with each one responding to proposals by others. Looking ahead to practical implementations, not just prototypes, this approach seems to offer some potential practical advantages, 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 this table to the code described in Section 2 is systematic. 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 of Table 1, and Figure 3.

From line 2 in the agent internal model, the code in Figure 3a 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 implicitly 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 3b 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.

### *3.2 Interagent Conversational Model*

The conversational model (Table 2) identifies the messages that must be exchanged for the agents to cooperate and negotiate with each other.

**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*) | PERMIT \| PROHIBIT | OCS |
| 3 | Prop(:connectFrom *a* :connectTo *b*) | PERMIT \| PROHIBIT | 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 |

Table 2 includes four types of messages, namely: Prop, CProp, PERMIT, and PROHIBIT. 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 (PERMIT), or proposal rejection (PROHIBIT). 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 2. Proposals correspond to tuples, and responses to proposal are implemented as tuples with status fields. The PERMIT and PROHIBIT messages are implemented respectively by the PERMIT and PROHIBIT statuses 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 6 summarizes the transformation of rows from the table expressing 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 to the corresponding row in the interagent conversational model.

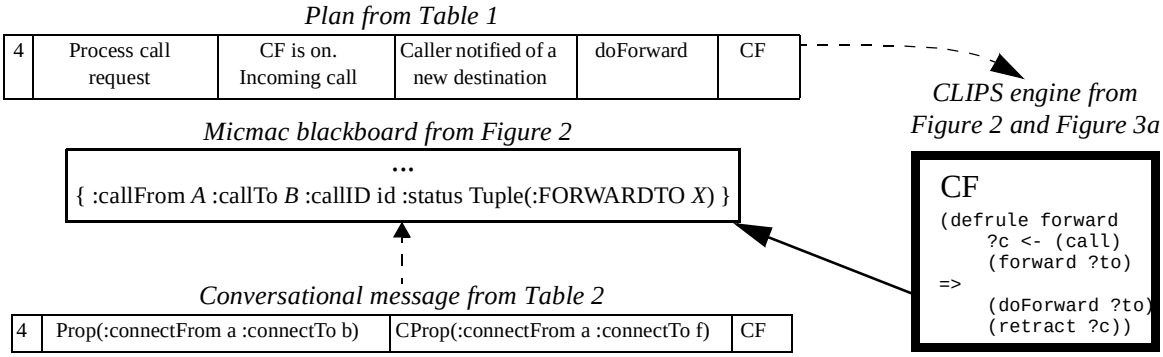


**Figure 6** From internal and conversational models to CLIPS rules and messages.

The conversation model describes protocols for direct negotiation between agents. The blackboard merely acts as an underlying medium, without reasoning about the progression of a negotiation. In its current preliminary form, this negotiation mechanism can be seen as a simplified version of the negotiation protocol discussed in [13]. However, we intend to extend it with the OPI model (Obligation-Permission-Interdiction) [2], which allows us to structure goals in a hierarchy of alternative, parallel, and sequential sub-goals. The OPI model is based on a deontic logic with formal propagation rules for its modalities (O, P, and I) and for acceptability. It allows for the automated reasoning about goal satisfaction, even in the presence of conflicting goals (this is solved with the notion of *cost* attached to modalities that are not satisfied). This model appears to be more expressive than the one in [13] because goals hierarchies can be used directly as messages for proposals (intentions), composite goals are of two kinds (parallel or sequential), agents involved in a negotiation might have different goal hierarchies (subscribed features), and the determination of acceptability is formal and decidable, even in the presence of costs attached to modalities.

## 4. High Level Agent Models from a Scenario-Path Notation

### *4.1 Scenario Paths for Agent Systems*

Scenario paths provide a means of representing the "structure of behaviour" for a whole system directly, in 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. Responsibilities are high-level activities that can be refined in terms of functions, tasks, procedures, events, and so forth. Composite diagrams that contain many possible paths (e.g., Figure 7), 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.

Particularly useful for feature interaction problems is the fact that UCMs are able to provide visual 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 (diamond-shaped) 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).

Figure 7 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 two additional features, 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, leading to system-wide paths that violate the intended behaviour of the features. The UCM allows a person to visualize the dynamic situations that give rise to feature interactions. The visualization is in terms that can be related to how agents are implemented.

Figure 7 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 another set 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), but is duplicated so the diagram can be read as suggested.

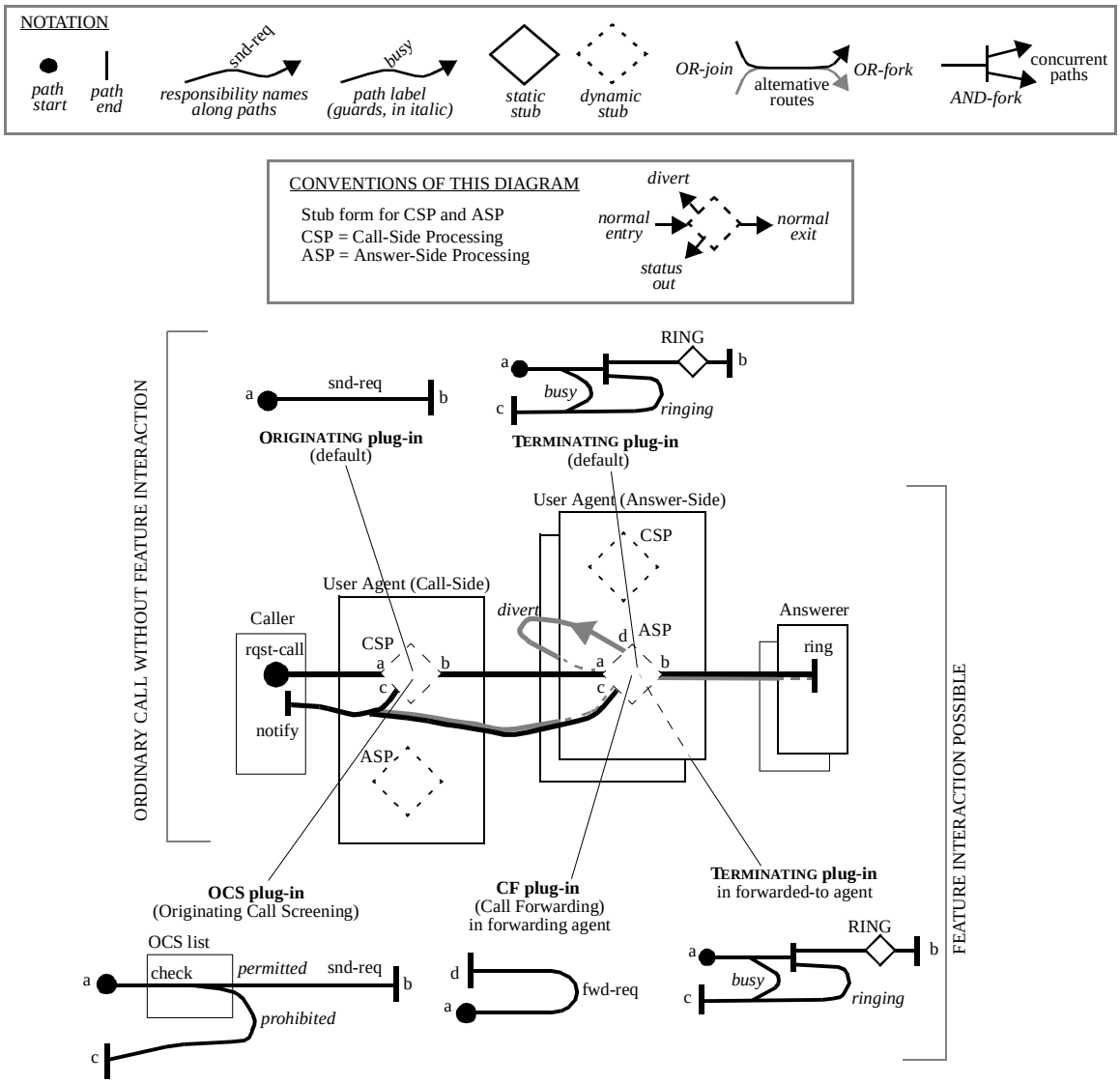


**Figure 7** A telephony feature-interaction example with a preliminary UCM.

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 another 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. Path labels (italicized on the map) represent conditions or guards attached to a particular path segment. Otherwise the scenario proceeds to an AND-fork. One path of this leads to a RING static stub (for which the unique plug-in is not 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 refuse, otherwise the caller is permitted to connect to the dialled number. This is shown on by the OR-fork and path labels that follow the *check* responsibility.

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 7 suggests 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 2 implements the different features as concurrent, competing rule engines that resolve the competition dynamically. In this case, the stubs are always active, and the UCM emphasizes the causal relationships between responsibilities, not necessarily the temporal relationships. Observe that UCMs make no commitment to which approach is taken.

### *4.2 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. The next two figures show trouble-free UCMs that result from doing this for Figure 7. In Figure 8, the OCS plug-in permits the call in agent *A*. Agent *B*'s CF plug-in is not activated, so the TERMINATING feature checks whether the answerer is busy, which is not the case here. In Figure 9, the OCS plug-in prohibits the call to *X*, and agent *A* is then notified accordingly. Figure 10 shows a feature-interaction-prone composition involving Originating Call Screening and Call Forwarding. This is a feature interaction if *X* is on *A*'s OCS list.

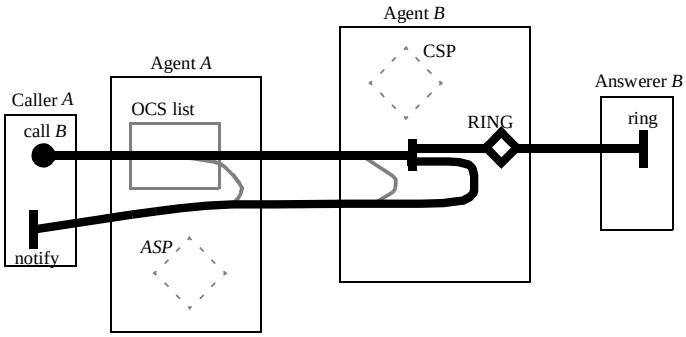


**Figure 8** Path view of "*A* connects to *B*" scenario.

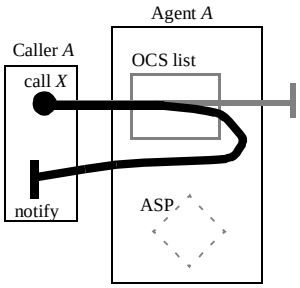


**Figure 9** Path view of "*A* refuses call to *X*" scenario.

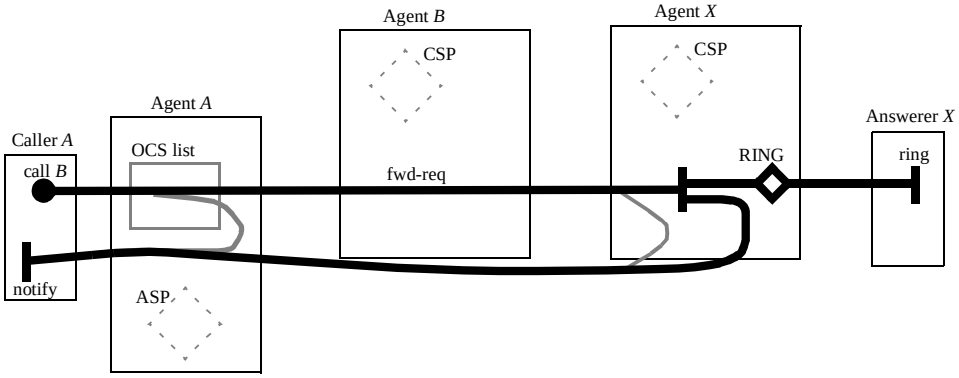


**Figure 10** Path view of "*A* allowed to call *X* scenario"
(a feature interaction if *X* is on *A*'s OCS list).

Figure 11 shows a different stubbed UCM (compare with Figure 7) 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. This is the UCM that was implemented in our prototype, not the one in Figure 7. We can observe that the paths in Figure 11 emerge from the rules and the execution scenarios in the simulation (Figure 3 and Figure 5). In general, redesigning a main UCM like this could require redesigning the plug-ins, just like adding new features might require redesigning the stubs and their context (and possibly agent heads and bodies).

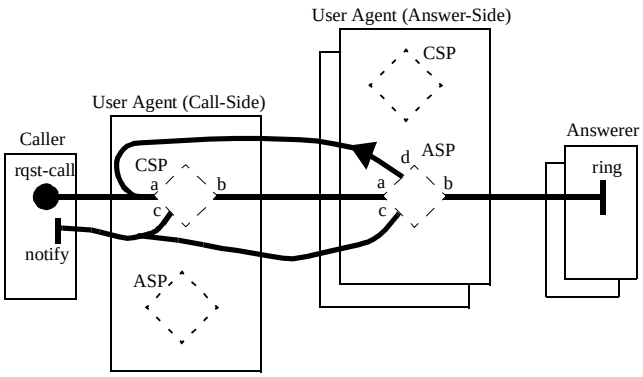


**Figure 11** Feature interaction resolution with a final UCM.

For several years, several academic research groups and industrial design teams have used UCMs in the design of real-time, distributed, and object-oriented systems. Their introduction into existing design processes was eased by the fact that people naturally use similar (although less precise and less formal) techniques for expressing scenarios visually with lines going through components [4]. We believe that UCMs can help a person to visualize, reason about, and resolve feature interaction problems in systems of agents at a high level of

abstraction, even before any commitment to design or implementation details such as messages, negotiations and architectures. 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.

### *4.3 From Use Case Maps to Intermediate Agent Models*

The nature of the relationship between UCMs and the intermediate agent models of Section 3 is summarized in Figure 12. Applying the mapping rules illustrated in Figure 12 to the UCM of Figure 7 leads to the agent internal model of Table 1. 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, using Figure 7 as the starting point requires that paths crossing the user agent in both roles must be mentally combined to get an agent-centric picture that covers all the possibilities expressed by the UCM.

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 relationship is defined by UCM paths and the causal *mechanism* is defined by the conversational model. In general, we visualize that different conversational models will be identified for different purposes, but this example does not illustrate this.

## 5. Discussion

### *5.1 Putting the Pieces Together*

The approach was 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).

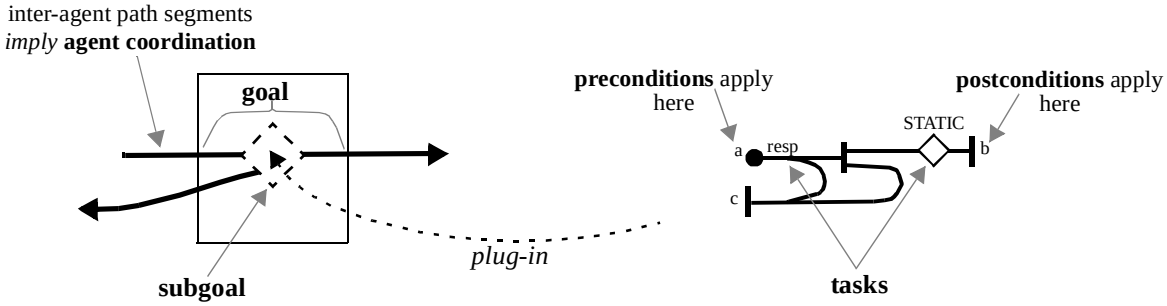


**Figure 12** From UCMs to agent internal and conversational models.

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.

UCMs do not *prove* that there could be feature interactions. They are intended to *visualize* potential interaction at an early stage of the design process. A more formal analysis of UCMs would require the use of formal languages such as OPI or LOTOS. We already developed several examples where UCMs have guided the drafting and validation of LOTOS specifications (e.g., [1]).

We are using this approach to investigate some difficult issues in the design and development of practical agent systems [8]. 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.

### *5.2 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 (the ones subscribed by individual users). 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 [16].

The rule selection strategy implemented in this prototype is based on local salience with fixed priorities. More flexible dynamic priority schemes are being considered. The authors are collaborating with Barbuceanu [2] and others to extend saliences into a system of dynamic priorities based on the OPI model, thus improving the flexibility and scalability of the approach.

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 generic languages to provide for component coordination (like KQML [17] and the FIPA [10] protocols) are changing assumptions about the lack of scalability and security 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 destroying localized ones when required. Distributing the blackboards would also augment the security and the computing power. Each enterprise could own its own local physical blackboard (with their own security policies), and each user could provide local computing power through a personal computer.

## 6. Conclusions

This paper shows by example how to apply a novel approach being developed for the systematic design and implementation of dynamic agencies to the problem of feature interaction. In this approach, Use Case Maps provide system-wide "behaviour structures" that enable people to get an early and global understanding of dynamic situations. Features are modelled as dynamic plug-ins for stubs in the UCMs. Feature interactions are seen visually and can be reasoned about and resolved by people at the UCM level. Tables generated from the UCMs provide a framework for humans to add information that will enable executable prototypes to be generated. These prototypes are FI-avoidant systems where features are modelled as competing rule engines and interactions are detected and resolved at run time by coordinating through a blackboard. The approach offers the promise of being scalable to practical numbers of features and of being practical for future commercial systems.


## Acknowledgments

This research is supported by Mitel, TRIO (now CITO), and NSERC. Contributions are gratefully acknowledged of A. Miga, G. Alexiu, D. Pinard, P. Perry, and M. Weiss.


## References


[1] D. Amyot, L. Logrippo, R.J.A. Buhr, *Spécification et conception de systèmes communicants : une approche rigoureuse basée sur des scénarios d'usage*, Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'97), Liège, Belgique, 1997. http://www.csi.uottawa.ca/~damyot/phd/cfip97/cfip97.html

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

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

[4] R.J.A. Buhr, *Use Case Maps as Architectural Entities for Complex Systems*, to appear in Transactions on Software Engineering, 1998. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/ucmUpdate.pdf

[5] R.J.A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, S. Mankovski, *High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature-Interaction Example*, Third Conference on Practical Application of Intelligent Agents and Multi-Agents (PAAM'98), London, UK, March 1998. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam98.pdf.

[6] 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.pdf

[7] 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.pdf

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

[9] E.J. Cameron, N.D. Griffeth, Y.-J. Lin, M.E. Nilson, W.K. Schnure, *A Feature Interaction Benchmark for IN and Beyond*, in Feature Interactions in Telecommunications Systems, IOS press, pp. 1-23, 1994.

[10] Foundation for Intelligent Physical Agents (FIPA). http://drogo.cselt.stet.it/fipa/

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

[12] J.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

[13] N.D. Griffeth, H. Velthuijsen, *The Negotiating Agents Approach to Runtime Feature Interaction Resolution*, in Feature Interactions in Telecommunications Systems, IOS press, pp. 217-235, 1994.

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

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

[16] 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), London, UK, 21-23 April 1997.

[17] University of Maryland Baltimore County, *UMBC Knowledge Query and Manipulation Language (KQML) Web*, 1998. http://www.cs.umbc.edu/kqml/

[18] 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.