VIPER: A VIsual Protocol EditoR

C.F.B. Rooney, R.W. Collier, G. M. P. O'Hare

Department of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland
{colm.rooney, rem.collier, gregory.ohare}@ucd.ie
http://www.cs.ucd.ie/

Abstract. Agent interactions play a crucial role in Multi-Agent Systems. Consequently graphical formalisms, such as Agent UML, have been adopted that allow agent developers to abstract away from implementation details and focus on the core aspects of such interactions. Agent Factory (AF) is a cohesive framework that supports the development and deployment of agent-oriented applications. This paper introduces the Visual Protocol Editor (VIPER), a graphical tool that allows users to diagrammatically construct agent interaction protocols. These protocols are subsequently realised through AF-APL, the purpose-built Agent-Oriented Programming language that sits at the heart of AF. In particular, we focus upon the design of interaction protocols using a subset of Agent UML. To this end, we specify a number of tools and an associated process through which developers can supplement these protocols with application- and domain-dependant AF-APL rules to create useful agents that adhere to the protocol constraints.

1 Introduction

Agent UML (AUML) is a language and set of graphical formalisms geared towards the design of agent oriented systems. It takes its primary inspiration from UML, a de facto standard for object-oriented design, but also draws from other agent design frameworks, e.g. MESSAGE [1], Gaia [17], Tropos [5]. One of the first areas of focus of the AUML community was the specification of graphical conventions for describing agent interaction protocols, i.e. sequence diagrams [10]. Due to the newness of the AUML effort however there is still little in the way of actual agent development tools that utilise AUML for specifying agent interactions. In [1] the authors of the MESSAGE system propose its use to enhance their interaction modelling phase and this is again reiterated in [12] which describes its extension IGENIAS. [8] describes a system that will generate code from AUML sequence diagrams. However, the code generated is pure Java and, as the author admits, is not well equipped to capture certain semantic elements of agent interactions and communication performatives. One approach to capturing such semantics, which we advocate in this paper, is through the use of an

Agent-Oriented Programming (AOP) language. An additional issue with the above solutions is that the source for the code generation process is a textual description of a protocol in a pre-defined notation and as such no graphical assistance is provided to the user. [15] has an early prototype of an AUML sequence diagramming tool that will take a textual description of a sequence diagram and display a graphical version. However once again the diagram may only be edited at the textual level. There are other agent development systems that actually provide diagramming tools for agent interactions that then link to code generators, e.g. AGIP [9], JiVE [4]. These however do not employ AUML and also generally produce code in common programming languages rather that more specialised AOP languages.

Therefore, this research aims to fill this gap by producing a suite of graphical tools that allow users to develop agent interaction protocols using the AUML conventions, which may then be realised as agent programs in an actual AOP language through a process of user-driven code generation. While a number of AOP languages exist, e.g. 3APL [6] and AgentSpeak(L) [14], we focus upon generating code for the Agent Factory Agent Programming Language (AF-APL), an AOP language that sits at the heart of the Agent Factory (AF) framework [2, 3, 11].

This framework delivers structured support for the development and deployment of applications comprised of agents that are: autonomous, situated, socially able, intentional, rational, and mobile [2]. To achieve this, AF combines: the AF-APL programming language; a distributed FIPA-compliant Run-Time Environment; an integrated development environment; and an associated software engineering methodology. A detailed explanation of AF is beyond the scope of this paper. Instead, we focus upon two aspects: AF-APL and the visual agent programming tools that are delivered as part of the Development Environment.

2 Agent Programming with AF-APL

AF-APL is a purpose-built AOP language that has been developed to support the fabrication of agents. As is usual for AOP languages, AF-APL agent programs combine an initial mental state (comprised of *beliefs* and *commitments*), with a set of rules that define the dynamics of that mental state (known as *commitment rules*). These are joined by a *plan library* that may be populated with a number of partial plans describing potential courses of action. The syntax and semantics of AF-APL is based on a logic of commitment, details of which may be found in [2].

Beliefs represent the current state of both the agent and its environment. In AF-APL, this state is realised as a set of facts that describe atomic information about the environment, and which are encoded as first-order structures wrapped within a belief operator (**BELIEF**). For example, in a mobile computing application an agent may be asked to monitor the users current position using a Global Positioning System (GPS) device. The agent may generate a belief about this position that takes the form: "**BELIEF**(userPosition(Lat, Long))" where Lat and Long are replaced by values for the users latitude and longitude respectively. In AF, the actual values for the latitude and

longitude are retrieved directly from the GPS device by a *perceptor unit*, which converts the raw sensor data into corresponding beliefs. The triggering of this perceptor unit is part of a perception process, which is central to our strategy for updating the beliefs of agents and is realised by triggering a pre-selected set of perceptor units at regular intervals. The specific set of perceptor units to be used by an agent is specified as part of the agent program through the **PERCEPTOR** keyword.

Commitments represent the courses of action that the agent has chosen to perform. That is, they represent the results of some reasoning process in which the agent makes a decision about how best to act. From this perspective, commitment implicitly represents the intentions of the agent. This contrasts with more traditional Belief Desire Intention approaches [14,16] in which intention is represented explicitly and commitment is an implicit feature of the agents' underlying reasoning process. This alternative treatment of commitment is motivated by our goal of explicitly representing the level of commitment the agent has to a chosen course of action. In AF-APL a commitment is comprised of: an agent identifier (the agent for whom the commitment has been made), a start time (before which the commitment should not be considered), a maintenance condition (which defines the conditions under which the commitment should not be dropped), and an activity (the course of action that the agent is committed to). Currently, an activity may take one of three forms: (1) an action identifier (i.e. some primitive action that the agent must perform), (2) a plan identifier (i.e. an identifier that can be used to retrieve a partial plan from the agents plan library), and (3) an explicit partial plan (i.e. partial plans can be directly encoded into a commitment).

Action identifiers are modelled as first-order structures where the parameters may be used to customise the action. For example, the activity of one agent informing another agent of something is realised through the "inform(?agent, ?content)" action. Here, the ?agent parameter refers to the agent to whom the message is to be sent (their identifier), and the ?content parameter refers to the content of the message. An example of an inform message can be seen in figure 2 below. Within AF-APL, actions are realised through the triggering of an associated *actuator unit*. As with perceptor units, actuator units are associated with specific agents as part of the agent program through the **ACTUATOR** keyword. Actions can be combined into plans that form more complex behaviours using one or more plan operators - currently there are four plan operators: sequence (**SEQ**), parallel (**PAR**), unordered choice (**OR**), and ordered choice (**XOR**). Plans can be stored within an agents internal plan library, where they are distinguished from one another by a unique plan identifier.

Finally, commitment rules describe the situations, encoded as a conjunction of positive and negative belief literals, under which the agent should adopt a given commitment. An implication operator (**=>**) delimits the situation and the commitment.

AF-APL agent programs (actuators + perceptors + plans + initial mental state + commitment rules) are executed upon a purpose-built agent interpreter. Specifically, the agent program is loaded into appropriate data structures inside the agent interpreter. The interpreter then manipulates these data structures through a simple control cycle that encapsulates various axioms defined in the associated logic of commitment. This cycle is comprised of three steps: (1) update the agents' beliefs, (2) manage the

agents' commitments, and (3) check whether or not to migrate. It is invoked repeatedly for the lifetime of the agent (at least whenever the agent is active).

```
// Perceptor & Actuator Configuration
PERCEPTOR ie.ucd.core.fipa.perceptor.MessagePerceptor;
ACTUATOR ie.ucd.core.fipa.actuator.InformActuator;
ACTUATOR ie.ucd.aflite.actuator.AdoptBeliefActuator;

// Initial Mental State
ALWAYS(BELIEF(providesService(docRelease)));

// Commitment Rules for DocService
BELIEF(fipaMessage(request, sender(?agt, ?addr), subscribe(?svc))) &
BELIEF(providesService(?svc)) =>
COMMIT(Self, Now, BELIEF(true), PAR(inform(?agt, subscribed(?svc)),
                    adoptBelief(ALWAYS(BELIEF(subscribed(?svc, ?agt))))));

BELIEF(newDocument(?doc)) & BELIEF(subscribed(docRelease, ?agt)) =>
COMMIT(Self, Now, BELIEF(true), inform(?agt, newDocument(?doc)));
```

Fig 1. An Example AF-APL program for a World Wide Web (WWW) spider agent.

By way of illustration, figure 2 presents a fragment of AF-APL code from a WWW spider agent that provides a service in which it informs subscribed agents of any new documents it finds. This is realised through two commitment rules and one initial belief. The first commitment rule states that if the agent receives a request to subscribe to a service (identified by the variable "?svc"), and the agent believes that it provides the service, then it should commit to performing two actions in parallel (specified by the **PAR** plan operator). The first action involves the agent informing the requester that they have successfully subscribed to the service, and the second action involves that adoption of a belief by the agent that the requester has been subscribed to the service. The second commitment rule states that if the agent believes that it has found a new document, and the agent believes that another agent (?agt) has subscribed to the "docRelease" service, then it should commit to informing that agent of the existence of the new document. Finally, the initial belief that the agent adopts on start up allows the agent to believe that it can provide the service "docRelease".

3 Applying AUML

A benefit of using a formal graphical language such as AUML for specifying agent interactions is that there is a solid underlying formal model that may be used to analyse and validate the interactions developed. This formal model also serves to constrain and guide any visual tools that support the language. As shall be seen in section 4, the system described in this paper uses such a model to manage how protocols are constructed. Each VIPER protocol diagram has an associated protocol model that stores the details of that protocol. The formal model serves to constrain what may be included in the protocol and thus we will refer to it as the *meta-model*.

In the terms of this work an interaction protocol diagram is regarded as comprising a set of visual components linked together by associations. For example in an AUML sequence diagram a lifeline is associated with an agent role [10]. Therefore, a meta-

model in this context is comprised of a set of component types, a set of association types and a set of constraints on how these components and associations relate. The following meta-model is used to provide a subset of Agent UML.

Components:

- ***Roles***. These are used to specify a particular agent instance, class or role acting within a protocol. They are represented graphically as rectangles containing text, see 1 in figure 2.
- ***Threads***. These represent the thread of control for particular roles within a protocol. Each thread corresponds to a section of an AUML sequence diagram lifeline and is represented as a vertical line, see 5 in figure 2.
- ***Messages***. These represent the asynchronous messages that make up the interaction protocols. They are sent as a result of and result in processing in Blocks. A message is described by a horizontal arrow (2 in figure 2).

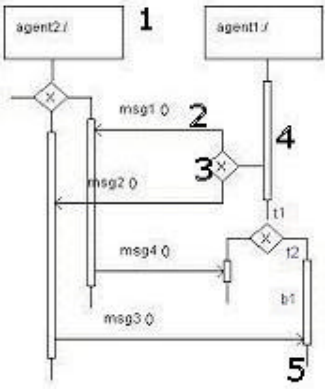

Fig 2. Dependant components

- ***Concurrent Guards***. These are used to represent a conditional guard on either a set of concurrent messages or concurrent lifelines. Three classes of guard are allowed: AND, OR and XOR. The component 3 in figure 2 illustrates an XOR guard.
- ***Blocks***. These represent the actual processing that takes place when sending and receiving messages in the protocol and are represented as rectangular blocks (see 4 in figure 2). When compiling agent designs from published protocols, it became apparent that there might need to be some constraints upon the associations between blocks and messages. For example, there was the issue of whether a block should be viewed externally as the change in state caused by sending a message, or internally as the processing that results in such. If we take the internal perspective, then any change in the ordering of messages within a block will have substantial consequences on any processing within that block, e.g. invalidating preconditions. Therefore, to facilitate agent code generation it was decided that the tool would implement a subset of AUML that enforced certain constraints on the associations between messages and blocks. The constraints were as follows:
  - If a block receives a message, then that message is received at the top of the block. That is, in terms of processing, the message receipt is the first act/event to be processed.
  - If a block sends a message, then that message is sent at the bottom of the block. That is, in terms of processing, the message sending is the final act/event to be processed.
  - A block may send at most one message and receive at most one message.
- ***Sub-blocks***. While block constraints solved some problems, they raised the issue of how to handle messages sent within blocks in the traditional AUML notation.

One such example is a request for clarification of an order as in figure 3. This convention was simply too useful to be omitted for the sake of implementation efficacy. This led to the introduction of the *sub-block* class of component. The sub-block is identical in almost every respect to a block with the exception that it may not be associated with any thread but rather is encapsulated within another block (or another sub-block). Furthermore the relationship between sub-blocks and messages is identical to that of blocks and messages. As such the insertion sequence example mentioned above could be represented using two sub-blocks: one to send the request for clarification (s1) and another to receive it (s2).

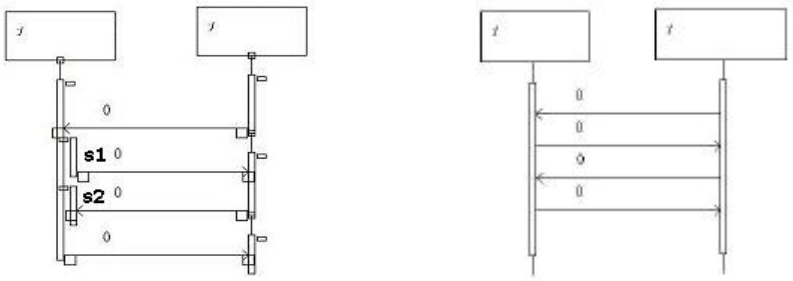

Fig 3. Use of sub-blocks.

- ***Null***. Another decision made with this version of the tool was that there be no un-associated components within the model. In order to capture the notion of floating fragments a *null component* has been introduced. Therefore, when a component is dis-associated from its parent component the newly un-associated component is associated with the null component instead.
- ***Root***. This is the parent component (in essence the canvas).

Associations:

- ***Dependency***. A dependant component is one that is related (directly or indirectly) to another component further down (temporally) that component's thread of control. For example, in figure 2, the block b2 is directly dependent on the thread t2 and they are both indirectly dependent on the thread t1. The SCA only keeps explicit track of associations between directly dependent components.
- ***Anchored***. When developing the meta-model it was decided that allowing components to be dependent on more than one parent component was unsuitable. As a result a message sent from a block in one role to a block in a different role is a dependant of the sending block alone and forms an anchored association with the receiving block. This captures the idea that the message belongs to the sender but also has some relationship with the receiver. Anchored associations are weaker than dependant associations and this is represented in the interface also. If a message is moved so that it is no longer in contact (or anchored) with the receiving block then the association is dissolved.

- ***Encapsulation.*** A third class of association allowed by the meta-model is encapsulation. This captures the relationship between blocks and sub-blocks.

Constraints*:*
The meta-model also defines the constraints that describe how the components are associated. For example, a component may only be dependant upon one other component, or a block may only receive one message. By altering these constraints the meta-model and hence the type of protocols it can represent can be customised.

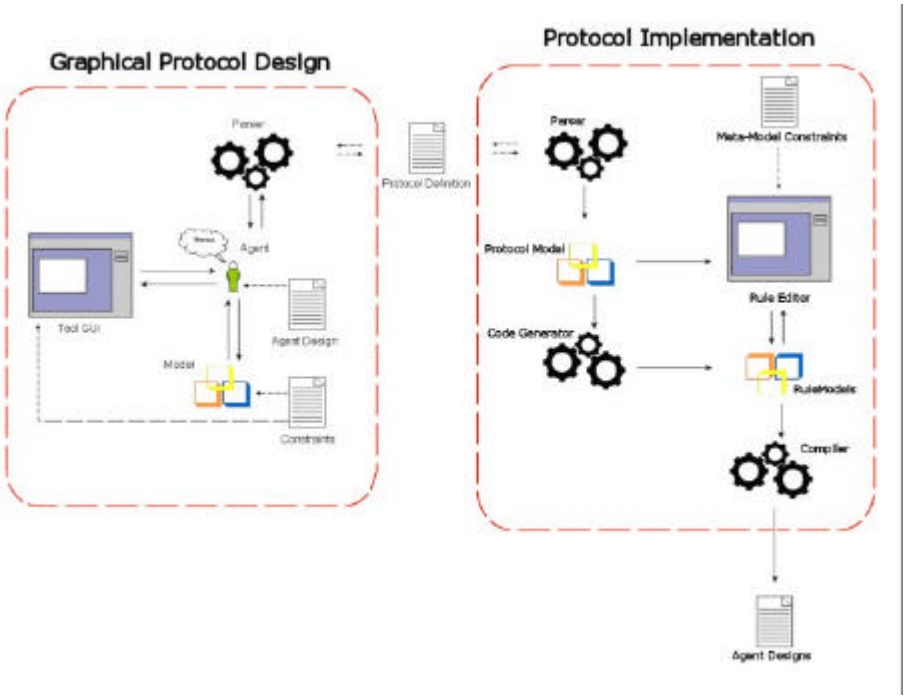

Fig 4. VIPER Agent Interaction Protocol Development Processes.

4 Visual Development of Agent Interaction Protocols

Visual development of agent interaction protocols within VIPER may be divided into two distinct processes as illustrated in figure 4. Firstly there is the design phase where a user graphically constructs a protocol diagram using the AUML sequence diagram. The second phase is implementation and involves the user populating the protocol with customised agent code, which together with code automatically generated to reflect the protocol semantics is compiled into useable agent designs.

4.1 Visual Design

The VIPER Protocol Editor (PE) is composed of three loosely coupled components: the Model, the GUI and the Semantic Checker Agent. This is analogous to a Model View Controller (MVC) architecture common in software engineering. The GUI is of course the front-end that the user sees and interacts with (see figure 6), the Model acts as the data store and the Semantic Checker Agent (SCA) implements the business logic. The flow of control is as follows: Any non-cosmetic user updates to the protocol (i.e. those that effect the Model) are forwarded from the Interface to the SCA. The SCA then queries the current state of the Model and evaluates the user's request in the light of this information and the constraints set down in the meta-model. If the request is successful the Model is updated accordingly and the updated details are communicated to the Interface component, which in turn displays the result to the user. In the event of a rejected request the Interface is also notified and can inform the user.

The system employs a component-based architecture with the flow of control between components being primarily event-based. One of the main design goals of this incarnation of the system was to make it generic and easy to re-configure[1]. By adopting such a component-based model the system is not as limited in terms of interchanging components. The only requirement is that the components implement a single method that allows them to receive the events. A wrapper class may even provide this functionality. The components are not required to respond to events, only those that hold special relevance to them and these can be handled at the component's leisure.

At the core of the system is the modelling language used to develop protocols. This defines what protocols may be developed and how. For a particular modelling language therefore the PE needs a meta-model (see section 3) in order to manage and constrain the Model. Within this work, an agent known as the SCA enforces the meta-model constraints. To achieve this the SCA is supplied with the following tools: (i) a Model Perceptor; (ii) a Tool Perceptor; (iii) various actuators for updating and querying the model, e.g. `addAssociation`, `requestOptions`, `saveProtocol`, and (iv) perceptors and actuators for parsing and interacting with the tool interface.

At every time step the model perceptor generates a set of beliefs that reflect the current state of the protocol model, i.e. the current elements and associations between them. The tool perceptor receives any user-requests made via the tool interface and translates them into beliefs that can then (in conjunction with the model beliefs) be used by the agent to decide the appropriate course of action. These decisions are guided by commitment rules (see figure 5) that have been written to enforce the semantic constraints laid down in the meta-model.

```
BELIEF(associate(?c1type,?c1,?c2type,?c2,?class)) &
BELIEF(comp(?c2,?c2type)) & !BELIEF(assocCount(?c2,?Assoc,?M)) &
BELIEF(legal(?Assoc,?c2type,?c1type,?N,?class))          =>
COMMIT(Self,Now,adoptBelief(BELIEF(addAssociation(?c1type,?c1,?c2,?class))))
```

Fig 5. Example Semantic Constraint Commitment Rule

[1] Inspired in part by the nascent state of the Agent UML specification.

Therefore, a developer would also be able to influence the tool behaviour and meta-model semantics by editing the agent definition, i.e. through their choice of perceptors, actuators and commitment rules.

4.2 Realising the Protocol: User-driven agent code generation

While a tool for the graphical design of protocols is useful as a visual aid, what the developer really needs in the end is agent code. The purpose therefore of the VIPER Rule Editor (RE) (see figure 7) is for the user to graphically associate rules with the various stages of a protocol. In order to achieve this they firstly need to be presented with a graphical representation of the protocol. To this end the RE has a canvas that displays the protocol diagram much as the PE has. As the user needs to interact with some of the protocol diagram elements, these too are stored in the canvas in the same fashion as the PE, with the exception that the protocol diagram is no longer editable.

The implementation process involves the following steps:

1. The protocol is loaded into the protocol model and an appropriate RuleSet subclass is instantiated for each editable component, e.g. blocks. The particular RuleSet subclass is defined in the RE configuration file. This file is analogous to the PE configuration file (section 6) and for every permitted component specifies the RuleSet subclass and an associated EditorPanel subclass.
2. The protocol model information is used to generate skeleton code (see figure 7) for the RuleSets.
3. The user fills in the blanks in this skeleton code. When a user selects a component, its associated EditorPanel is displayed. This is a view on the underlying RuleSet and allows the user to review and manipulate any skeleton and/or user-generated code stored in the RuleSet.
4. When the user has finished editing the code they choose the *compile* option. The RE then divides the protocol into the various roles and each role is compiled from those RuleSets associated with its dependent components.

The following mappings are used in the skeleton code generation process:

- ***Blocks*** are mapped to Agent Factory plans. The user fills in the blanks by entering actions or other plan names combined using the plan operators (section 2).
- ***Messages*** are in essence the pre- and post-conditions for the Block RuleSets. However, in order to remove agents from the burden of storing every message received and sent, the code generation process translates a message received/sent event to a unique landmark. As these landmarks are updated by each event, the agent need only retain the most recent landmark as that adequately reflects the current state of the protocol. Note that as these landmarks act as the link between protocol RuleSets the user is responsible for ensuring that any code they add does not interfere with landmarks or their ordering. This may involve the user having to add their own intermediate landmarks and suitable bridging code.

- ***Guards*** for concurrent messages or threads are mapped to a set of commitment rules, one for each subsequent branch. These rules take the form of a pre-condition with the previous landmark and a commitment to generate a landmark that uniquely identifies the branch. For each rule, the user then adds any other pre-condition terms needed to represent the guard.

5 Case Study

In order to highlight the operation and usage of VIPER we commission a simplified FIPA query-if protocol as illustrated in figure 6. As the user edits the protocol diagram the SCA constantly monitors their progress and guides them through the process. For example, whenever the user selects a component the SCA is notified and uses the information to provide an *adaptive interface* via contextualised menus. In terms of this example therefore, when the user starts with a blank canvas (i.e. the current component is the root component) the SCA updates the menus so that the user is only presented with the option to add a role component.

When a user chooses to add a role (or any component) the following steps take place. Firstly, the interface generates an event that the user wishes to associate a new role with the current component (i.e. the root canvas). Note that from the interface perspective all updates on the model take the form of adding, updating or removing associations. If the component to be associated does not yet exist then it is created by the SCA. Once the event has been generated it is picked up by the SCA Tool Perceptor, which adds it to the SCA's belief set. At the next deliberation cycle this belief about the association request fires commitment rules that have the effect of checking the feasibility of the user's request against the current state of the protocol model. Factors taken into account are: the type of association; the type of components; any existing associations involving either component; and the maximum associations of the requested type allowed for these components. As a result of this deliberation the user's request is either permitted or rejected and the SCA's belief set is updated to reflect this. In the case where the request is granted then the resulting belief causes the agent to fire commitment rules that update the model and notify the tool interface (via events) of the update in the protocol. The interface then uses this information to update the graphical display, i.e. in this case adding the new role to the canvas. If however, the user's request is refused, then the SCA sends a corresponding error event to the interface. This event can be used by the interface to display a contextualised error message to the user.

Once a role component has been added the user can then edit its details by selecting it and entering details in the panel (specified in the configuration file) that appears at the side of the GUI see figure 6. If the user then reselects the canvas they are presented with the option to add additional roles. In this way we can insert the roles needed for our example protocol as in figure 6.

Having established our roles the next step is to add lifelines and start to build the protocol in earnest. When a user selects a role once again the SCA uses that knowl-

edge to contextualise the GUI. In this case the user is presented with the option to add a thread to the role. Once a lifeline is in place the user can add a block. As described in section 3, blocks represent computation within the protocol and result from and/or result in the sending of a message. Therefore, when a block has been added the user is free to add a message sent from that block, or to associate a message received with the block. The latter is achieved by dragging the arrow of an existing message sent from another block so that it intersects the block the user wishes to receive the message. This triggers an association event and a process analogous to the one described above. As a result the SCA either permits or denies the association formation. In either case the interface displays the result to the user.

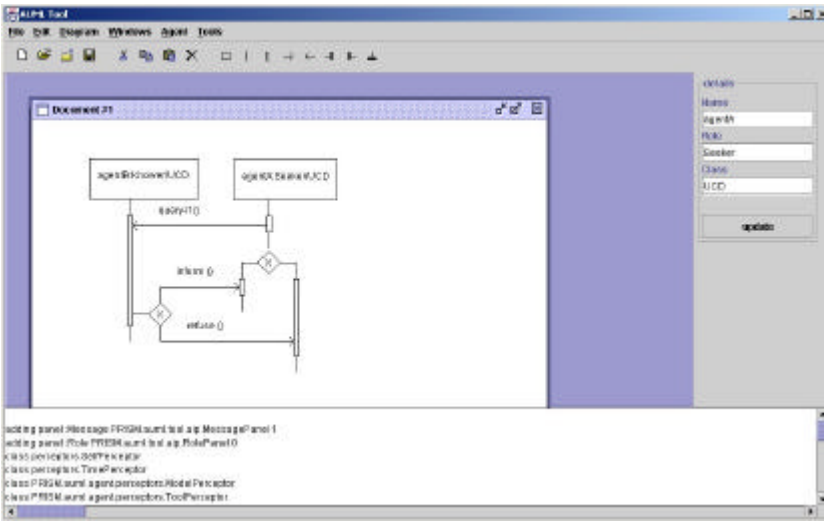

Fig 6. Protocol Design Interface

To continue the protocol and begin a new block of computation the user selects the final block along the given lifeline and adds a new thread after it and then a new block to that thread. The tool also incorporates features for deleting and dis-/re-associating (groups of) components. In this way diagrams can be chopped up and re-organised as needed, as long as the result is deemed semantically consistent by the SCA. (Groups of) Dis-associated components are referred to as *floating fragments* and are associated with the null component by the SCA so as not to corrupt the model.

When the protocol diagram is finished the user must save it before proceeding to the next phase of development. This involves the interface notifying the SCA of the filename chosen by the user and the SCA then using this to fire an actuator that outputs the model data to the appropriate file. In actuality there are two files created. It was decided that rather than a single file with semantic and display information combined, a better solution would be to decouple the graphical display information from the actual protocol representation. This separation of concerns means that if the format of one changes it does not affect the other. While protocol diagrams are graphical

in nature, this is really just a convenience to assist developers. The real power of them is the actual protocol being represented, i.e. what messages are sent by and to whom and when. This information should be independent of how exactly it is visually represented. The first of these files is the Protocol Diagram Definition file that contains the information needed to build up the protocol model, i.e. components and associations. The second file is the coordinates file. This file stores the coordinates and dimensions of each component in the diagram and is used to position them when the protocol is reloaded.

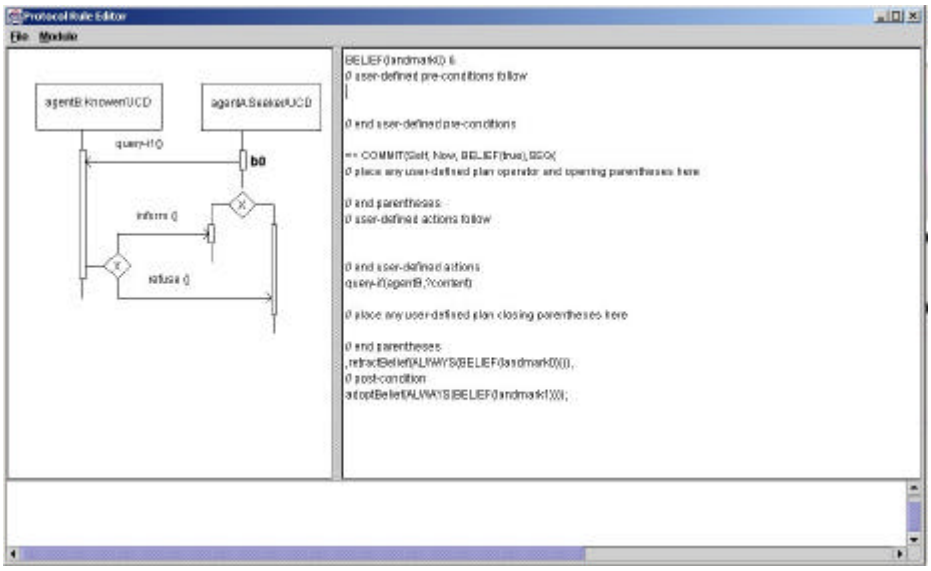

Fig 7. VIPER Rule Editor Interface

Once the user has saved a version of the protocol diagram they are happy with they can begin the next phase of the development process. As described in section 4.2, the RE is used to graphically associate rule sets with the blocks of computation in a protocol diagram. The code generator utilises the meta-model mappings detailed in section 3 and the protocol model information to generation the agent skeleton code. This code is stored in the RuleSets and is used to frame the user-defined rules within the constraints of the protocol diagram. Whenever a component is selected in the RE, the information in the corresponding RuleSet is loaded and displayed. Continuing with our example, the block of computation (b0) represented in figure 7 would result in the illustrated skeleton code.

The user is now free to add their agent code to the designs. In order to do so they select the component they wish to ascribe functionality to and use the displayed EditorPanel to edit the code. For example, in figure 7 block b0 has an associated EditorPanel subclass that displays the agent code using a text-editor interface. A user could add additional pre-conditions to determine when to send the query-if or include additional activities to be achieved before the message is sent. The pre-conditions are

specified using the AF-APL **BELIEF** construct. Activities are either primitive actions or plans as described in section 2. The user also needs to explicitly specify a message content for the placeholder `?content` or include pre-conditions that bind a value to it.

When the user is ready, the final stage of the process is the compilation of the actual agent definition files. The compiler splits the combines the RuleModels into files based on the roles they belong to in the protocol. Agent Factory users may then compose these files into workable agent designs.

6 Discussion and Conclusions

In an attempt to increase system flexibility we utilize configuration files, rather than hard-coding features and functionality. Two files are used to initialise the protocol design system: the *VIPER PE configuration file* and the *agent design*. The latter contains an AF-APL agent program (as specified in section 2), which is used to instantiate the SCA. The former is used in configuring both the tool interface and the model. It details in XML the types of components that may be added to the model and consequently displayed by the interface. The following information is detailed for each component type: (i) the name used to represent this type of component; (ii) the Java class that implements the graphical component - this must be sub-classed from the `ModelElement` class; (iii) any Java methods (and their corresponding arguments) that are needed to initialise the component; (iv) the icon file (if any) to be used for menus and buttons in the tool; (v) the event string to be fired when a new component of that type is requested; (vi) the Java panel class (if any) that is used in the tool to display and edit any component specific information, e.g. message string and performative; (vii) and any triggers that the component will need.

Using this information the tool can load component classes (using Java Reflection API) and recognise the component events without them having to be hard-coded. In this way simply editing the configuration file can change the type of diagram that may be edited. The XML fragment below illustrates how a component is represented in the constraints definition file.

```
<COMPONENT name='Message'>
     <CLASS name='MessageComponent' package='PRISM.auml.tool.aip'>
          <INIT><METHOD name='setType'><ARG value='Message'/></METHOD></INIT>
    </CLASS>
    <ICON name='arrow1.gif' path='classes\\images\\'/>
    <EVENT type='dependency'/>
    <PANEL><CLASS name='MessagePanel' package='PRISM.auml.tool.aip'/></PANEL>
    <TRIGGER><CLASS name='DependantMessageTrigger' package='PRISM.auml.tool'/></TRIGGER>
    <TRIGGER><CLASS name='AnchoredMessageTrigger' package='PRISM.auml.tool'/></TRIGGER>
</COMPONENT>
```

Upon parsing some of the details from this file is also fed in to the SCA. The SCA makes use of this information to build up the list of the components it can support and any constraints related to them, e.g. the number of messages a control block can send. In addition to component information, the configuration file also defines the types of associations permissible by the meta-model. This (in conjunction with the component

information) allows a designer to alter the semantics of the model by simply editing the configuration file. A typical association would be represented in XML as below.

```
<ASSOCIATION class='dependency' type='output' comp1='Block' comp2='Thread' max='1'/>
```

This configurability will play a large part in the evolution of the tool. The iteration of the tool described here is based upon AUML sequence diagrams as they are presented in [10] and as such does not incorporate the newer features proposed in [7]. In addition, as mentioned above, the meta-model only supports a subset of AUML. Certain features were omitted in this prototype in order to simplify the meta-model, ensuring a focus could be placed on determining the links between firstly the meta-model and the graphical model and secondly the protocol model and the agent design. So, for example, features such as synchronous messages while supported graphically are not supported by the meta-model as yet.

The next step in the development of the tool will involve augmenting the meta-model to incorporate a fuller range of AUML features e.g. some of the newer features proposed in [7] could play a very important role in the process of implementing protocols using the RE. In particular, AUML sequence diagram constraints and actions would allow protocol designers to include more implementation information within protocol diagrams. However, it should be noted that in [7] there is no fixed format in which these constraints and actions are to be specified. Therefore, automatically generating skeleton code would be impossible unless we constrained the VIPER Design Tool to only accept certain supported formats. If another format were used then the onus would be on the developer to fill in the blanks.

This paper has introduced VIPER, a visual protocol editor, which has been successfully integrated into Agent Factory (AF). It provides partial tool support for the design phase of the AF Development Methodology. While VIPER automatically generates agent code skeletons, necessarily, the designer will need to customise fragments within these. Notwithstanding this, VIPER surpasses previous visual tools for protocol editing in that code generation is more complete; it provides for richer expression of semantic elements of agent interactions; it enforces a separation of concerns between the editing tool, the underlying semantic model, and the agent implementation framework; and finally, it provides an adaptive user interface. VIPER currently provides the most complete AUML sequence diagram editing and code generation tool.

References

1. Caire, G., Leal, F., Chainho, P., Evans, R., Garijo, F., Gomez, J., Pavon, G., Kearney, P., Stark, J. & Massonet, P., Agent Oriented Analysis using MESSAGE/UML, in proceedings of the International Workshop Series on Agent-Oriented Software Engineering (AOSE), (2001).
2. Collier, R.: Agent Factory: A Framework for the Engineering of Agent-Oriented Applications, Ph.D. Thesis, Department of Computer Science, University College Dublin, Ireland (2001).

3. Collier, R.W., O'Hare G.M.P., Lowen, T., Rooney, C.F.B., Beyond Prototyping in the Factory of the Agents, 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03), Prague, Czech Republic (2003).
4. Galan, A.K., JiVE: JAFMAS integrated Visual Environment, MSc Thesis, Department of Electrical and Computer Engineering and Computer Science of the College of Engineering, University of Cincinnati, (2000).
5. Giorgini, P., Kolp, M., Mylopoulos, J., and Pistore M., The Tropos Methodology: an overview, in F. Bergenti, M.-P. Gleizes and F. Zambonelli (Eds) Methodologies And Software Engineering For Agent Systems, Kluwer Academic Publishing (New York), December (2003).
6. Hindriks, K.V., de Boer, F.S., van der Hoek, W., and Meyer, J-J., Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems, 2(4):357-401, (1999).
7. Huget, M-P., Bauer, B., Odell, J., Levy, R., Turci, P., Cervenka, R., and Zhu, H., FIPA Modelling; Interaction Diagrams, FIPA Working Draft, (2002).
8. Huget, M-P, Generating Code for Agent UML Protocol Diagrams, In Proceedings of Agent Technology and Software Engineering (AgeS), Bernhard Bauer, Klaus Fischer, Jorg Muller and Bernhard Rumpe (eds.), Erfurt, Germany, October (2002).
9. Koning, J-L, AGIP: a tool for automating the generation of conversation policies, in Z. Shi, editor, Proceedings of 16th IFIP World Computer Congress, Intelligent Information Processing (IIP-00), Beijing, China, August. (2000).
10. Odell, J., Van Dyke Parunak, H., Bauer, B., Representing Agent Interaction Protocols in UML, Agent-Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge eds., Springer, Berlin, pp. 121-140, (2001).
11. O'Hare, G.M.P.: Agent Factory: An Environment for the Fabrication of Distributed Artificial Systems, in O'Hare, G.M.P. and Jennings, N.R.(Eds.), Foundations of Distributed Artificial Intelligence, Sixth Generation Computer Series, Wiley Interscience Pubs, New York (1996).
12. Pavon, J., Gomez-Sanz, J., Agent Oriented Software Engineering with INGENIAS, 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03), Prague, Czech Republic (2003).
13. Rao, A.S. and Georgeff, M.P., Modeling Rational Agents within a BDI Architecture, in Proceedings of Second International Conference on Principles of Knowledge Representation and Reasoning, (J.Allen, R.Fikes, and E.Sandwall eds) pp 473-484, Morgan-Kaufmann, San Mateo, CA, (1991).
14. Rao, A., AgentSpeak(L): BDI Agents speak out in a logical computable language, in Proceeding of the 7th International Workshop on Modelling Autonomous Agents in a Multi-Agent World (de Velde, W. Va;Perram, J. W eds) , Eindhoven, The Netherlands, January 22-25 1996. LNAI 1038. Springer Verlag, (1996).
15. Winkoff, M., http://goanna.cs.rmit.edu.au/~winikoff/auml/index.html, (2003)
16. Wooldridge, M., Practical Reasoning with Procedural Knowledge: A Logic of BDI Agents with Know-How, in Proceedings of the International Conference on Formal and Applied Practical Reasoning, (D. M. Gabbay and H.-J. Ohlbach, eds), Springer-Verlag, (1996).
17. Wooldridge, M., Jennings, N.R., and Kinny, D., The Gaia Methodology for Agent-Oriented Analysis and Design, in Journal of Autonomous Agents and Multi-Agent Systems 3 (3) 285-312, (2000).