Defining syntax and providing tool support for Agent UML using a textual notation

Michael Winikoff

School of Computer Science and Information Technology
RMIT University, Melbourne, Australia
E-mail: michael.winikoff@rmit.edu.au

Abstract: An important role in software engineering is played by design notations. The Agent UML (AUML) notation for sequence diagrams has been widely used to capture the design of interactions between agents. However, AUML is not precisely defined, and there is very little in the way of tool support available. We argue that using a textual notation allows the notation to be precisely defined, and facilitates the development of tool support. We present a textual notation that we have developed, and describe a number of tools that support this notation. One of these tools is a 'renderer' which takes a textual AUML protocol and generates the standard graphical view. The layout of graphical elements in the generated graphical view is done automatically, using a layout algorithm which we present.

Keywords: agent-oriented software engineering; Agent UML; AUML; interaction design; design notations.

Reference to this paper should be made as follows: Winikoff, M. (2007) 'Defining syntax and providing tool support for Agent UML using a textual notation', *Int. J. Agent-Oriented Software Engineering*, Vol. 1, No. 2, pp.123–144.

Biographical notes: Michael Winikoff is an Associate Professor at RMIT University. His research interests concern notations for specifying and constructing software. In particular, he is interested in agent-oriented software engineering methodologies and is coauthor of the book *Developing Intelligent Agent Systems: A Practical Guide*, published by John Wiley and Sons in 2004.

1 Introduction

A crucial role in the design of software is played by various *notations* which are used to capture aspects of the design. The *de facto* standard notation for object-oriented design is the Unified Modelling Language (UML).[1] However, in the design of agent systems there are not yet widely accepted standard notations for design: methodologies such as Prometheus (Padgham and Winikoff, 2004), Tropos (Bresciani *et al.*, 2004), Gaia (Jennings *et al.*, 2004), MaSE (DeLoach *et al.*, 2001), Roadmap (Juan *et al.*, 2002), and many others (Bergenti *et al.*, 2004; Henderson-Sellers and Giorgini, 2005) each have their own notations. One proposed notation that has found a reasonable level of community acceptance is Agent UML (AUML).[2] Although AUML includes a number of notations (sequence diagrams, interaction overview diagrams, communication diagrams and timing diagrams (Huget and Odell, 2004)), its *sequence diagram* notation has been the most influential, having been adopted by a number of methodologies (*e.g.*, Prometheus, Gaia and Tropos) for describing agent interactions, and by FIPA[3] for describing standardised protocols.

Like most software design notations AUML is graphical. There are many reasons for this including readability and intuitiveness, however, there are also some drawbacks to graphical notations. Firstly, it is harder to precisely define the syntax of graphical notations, resulting in such notations often being defined by examples and thus lacking a precise and complete formal syntax. Secondly, it requires considerably more work to provide tool support. AUML suffers from both these problems: it is not precisely defined, and there is currently a lack of adequate tool support.

This paper argues that these problems can be addressed by using a *textual* notation for AUML. This is possible (and, in fact, quite natural) because the newer version of AUML (Huget and Odell, 2004) is much more structured than the previous version. By using a textual notation we are able to provide a precise, formal and simple definition of the syntax of AUML sequence diagrams.

Textual notations are also easier to support with tools. This is demonstrated by the current tools: there are a few tools that support AUML, and these are still in early stages, and are limited in terms of the features of AUML that they support. On the other hand, we are able to describe *four* tools which provide (different) support using the textual notation. The original AUML rendering tool takes the textual notation and produces a graphical rendition. A newer tool, which was developed by an undergraduate student over the summer vacation, extends UMLet[4] with support for (a variant of) the textual notation. Support for AUML, using the textual notation described here, has also been added to the Prometheus Design Tool (PDT)[5] (Padgham *et al.*, 2005a). Finally, the textual notation can also serve as an interchange format between tools: it is easy to both generate and to read. Indeed, the tool of Casella and Mascardi for converting AUML to WS-BPEL (Casella and Mascardi, 2006) is able to generate output in the textual format that we defined.

We have argued that by using textual notations we can address the lack of precise syntax, and can simplify the provision of tool support. However, we have also found that the textual notation can be used in its own right to notate interaction protocols. It is widely believed that design notations ought to be graphical, but we have found that the textual notation is quite usable. This was not only the experience of the author, but the AUML tool described has been made available to undergraduate students in a course covering agent oriented programming and design. The students were able to use the notation and tool effectively with very limited training (indeed, significantly more time was spent explaining the meaning of AUML constructs than was spent on explaining the textual notation or the tool).

Textual notations also have a number of other advantages: text editors provide a range of editing functionalities (find and replace, copy and paste, *etc.*) that are not generally provided by current graphical tools, and it is easy to leverage various existing tools such as version control. Furthermore, textual notations encourage the use of light-weight tools and encourage the developer to focus on the logic, not the appearance, of the design (Spinellis, 2003). Finally, using the textual notation it is very easy to automate various tasks such as finding all agents that communicate with a given agent, finding all protocols in a design that use a particular message type or sub-protocol, *etc.*

More generally it has been argued that textual notations provide insurance against obsolescence, leverage of existing tools (from version control to editors to compilers), and easier testing (Hunt and Thomas, 2000, Chap. 3); and that textual formats are preferable because of interoperability, transparency, and extensibility (Raymond, 2004, Chap. 5).

Despite these advantages, we do not argue that AUML ought to be written using the textual notation. Although we have found this easy and practical, we believe that people will continue to want to use the graphical notation. We thus view the textual notation as an alternative to the graphical presentation.

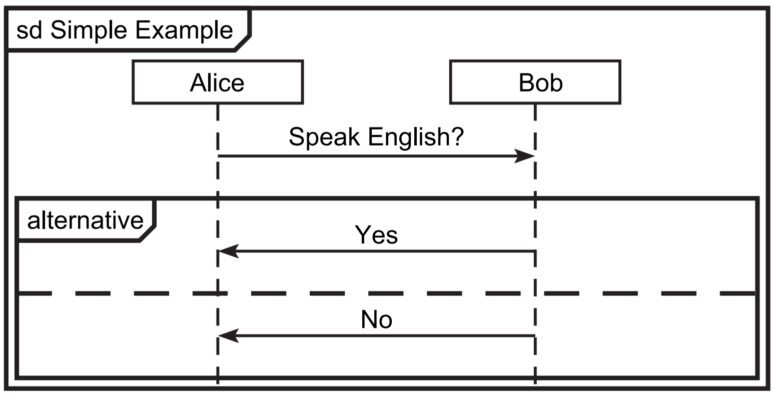
Compared with the earlier paper (Winikoff, 2005), this paper includes additional features in the textual notation, describes additional tool support, and includes a description of the layout algorithm used to render the textual notation into the standard graphical depiction of AUML.

This paper is structured as follows. We begin with a brief description of the AUML notation (Section 2). We then present a precise definition of a (subset of) AUML using a textual notation (Section 3), followed by a description of tools that utilise this notation (Section 4). We then discuss related work (Section 5) and conclude (Section 6).

2 The AUML notation for protocols

The AUML notation for protocols is similar to version 2.0 of the UML (OMG, 2003). An interaction protocol ('sequence diagram') consists of a number of lifelines, each labelled with an agent class name[6] in a box at the top of the lifeline. Messages are depicted by labelled arrows between lifelines, and time increases down the page. For example, in Figure 1 there are two agent types, called 'Alice' and 'Bob', and the first message that is sent is 'Speak English?' from Alice to Bob. Whether a message is synchronous or asynchronous is depicted using a different arrowhead: a synchronous message has a filled arrowhead, whereas an asynchronous message has an empty arrowhead. In Figure 1 all messages are synchronous.

Figure 1 A simple AUML protocol



Agent UML allows for alternatives, parallelism and so on to be specified using *boxes*. A box surrounds a part of the sequence diagram and has a type such as ‘Alternative’, ‘Option’, or ‘Parallel’ (given in the top-left corner of the box). Boxes can contain AUML elements such as messages and other boxes, *i.e.*, they can be nested. Boxes can also be divided into a number of *regions*, separated from each other by heavy horizontal dashed lines. Each region can contain a guard, depicted as text in square brackets, specifying a condition on that region being selected. For example, in Figure 1 there is an *alternative* box with two regions. The first region contains a message (labelled ‘Yes’) and the second region contains a message (labelled ‘No’). An example of nested boxes can be seen in Figure 4, which also shows a box with a guard (the *option* box).

AUML defines a number of box types including Alternative, Option, Break, Parallel and Loop:

- Alternative – Exactly one[7] of the box’s regions is executed. For example, the simple protocol in Figure 1 shows a message from an agent of type Alice to an agent of type Bob followed by *either* a reply of ‘Yes’, or a reply of ‘No’.
- Option – This box type can only have a single region and specifies that this region may or may not occur. It is equivalent to an alternative box with a second, empty, region.
- Break – Terminates the interaction. It isn’t entirely clear from (Huget and Odell, 2004) whether other threads of the interaction are terminated at the start of the break box or at the end.
- Parallel – Specifies that each of the regions takes place in parallel and that the sequence of messages is interleaved.
- Loop – Can only have a single region. Specifies that the region is repeated some number of times. The tag gives the type (‘Loop’) and also (optionally) an indication of the number of repetitions which can be a fixed number (or a range) or a Boolean condition (OMG, 2003, p.413). By default ‘Loop’ on its own means ‘zero or more times’.

In addition to these box types (and others, *e.g.*, critical region) AUML provides a number of other constructs (see Figure 3):

- Ref – This box type is a little different in that it does not contain other AUML elements, only the name of a protocol. The interpretation of the Ref box is obtained by replacing it with the protocol it refers to.
- Continuations – Incoming and outgoing continuations are just (respectively) labels and gotos. Both continuations are depicted by rounded rectangles; outgoing continuations (goto) have a right pointing triangle on their right side whereas incoming continuations (labels) have a right pointing triangle on their left side. Each goto must have a single corresponding label in the protocol, and when a goto is reached execution continues at the corresponding label.
- Stop – Depicted as an X on the lifeline of an agent, this denotes ‘the end of participation of a lifeline in the communication’ (Huget and Odell, 2004, Section 2.9).

- Timing constraints – Timing constraints can be depicted in a number of ways. One way is to indicate a region of the sequence diagram (see Figure 2) and give a constraint on the time. For example, the constraint in Figure 2 states that the response must be received within seven days of the request. An alternative way of depicting timing constraints (not supported by the textual notation) is to attach an indication of the form 't = now' to a message, and then later use indications such as '{t ... t + 7d}'.
- Crossed messages – Depending on the transport mechanism used for messages, it can be possible for (asynchronous) messages to arrive out of order. AUML provides a way of depicting this (see Figure 6). Although this is arguably more useful for traces than for protocols, AUML provides this, and so we support the notation.

Figure 2 Example of timing constraint

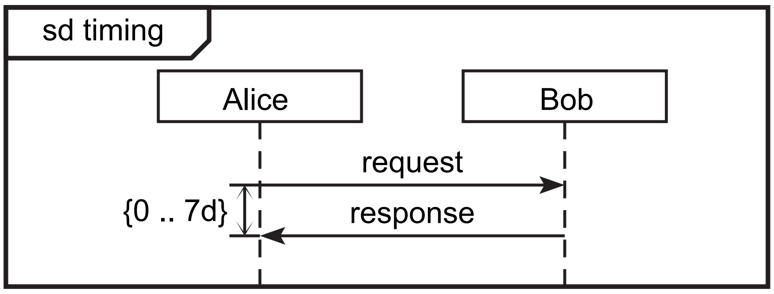


Figure 3 Additional AUML notation

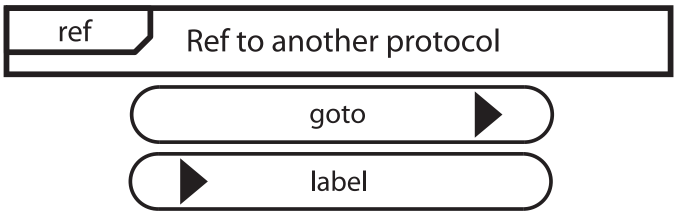
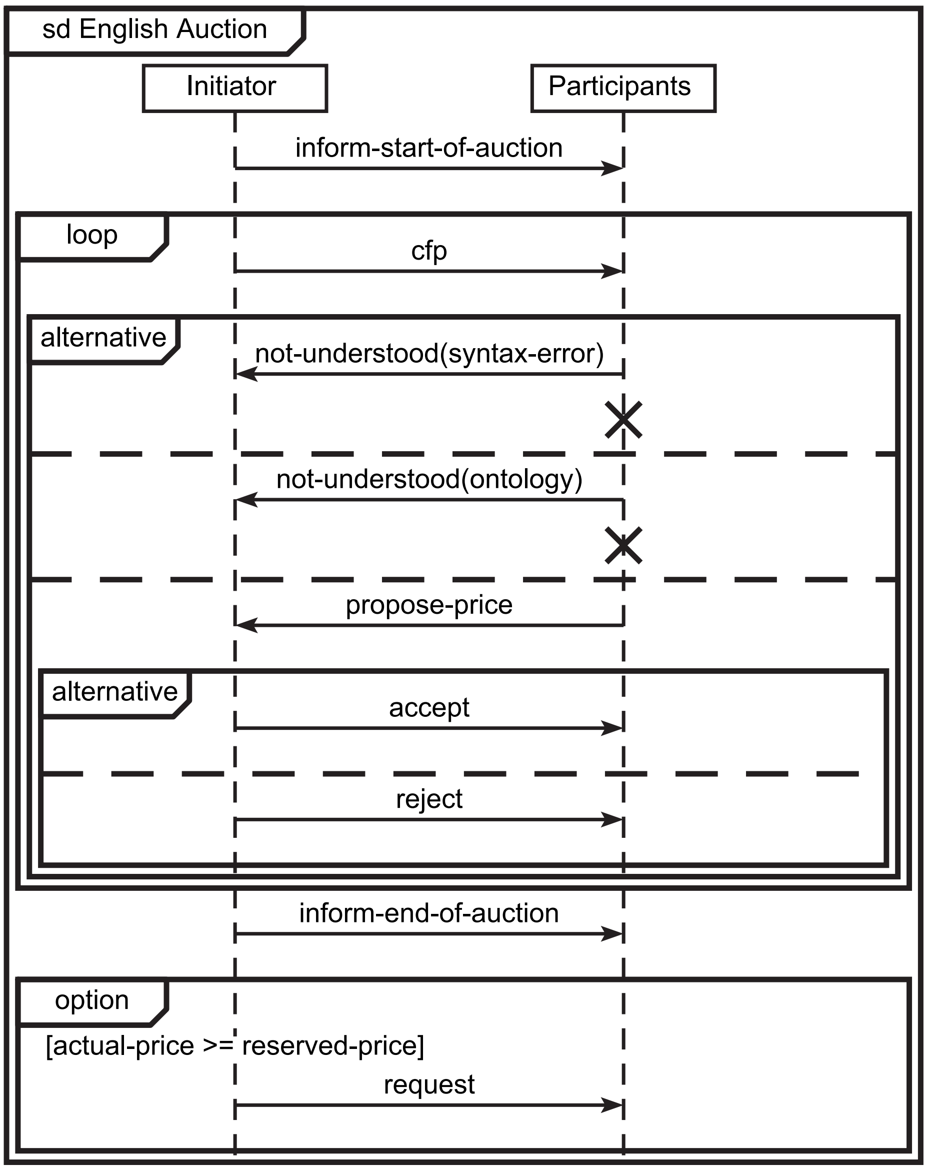


Figure 4 shows an English auction protocol. The protocol begins with the Initiator informing the Participants of the start of the auction. This is then followed by a number of rounds each of which consists of a call for proposals (cfp) from the Initiator to Participants, followed by a response from the Participant. If a Participant's response is not understood, then that participant does not continue to interact in the protocol. Otherwise, the response is a price (propose-price) which is accepted or rejected by the Initiator. Finally, the Initiator informs the Participants that the auction has finished (inform-end-of-auction), and if the price reached exceeds or equals the reserved-price then the Initiator requests the item.

Figure 4 English auction protocol



3 A textual format for AUML

A textual AUML protocol (see Figure 7 for the formal grammar) consists of a sequence of commands (one per line). The first line defines the name of the protocol (`start` *name*) and the last concludes the protocol (`finish`). Commands in between are used to:

- Define agents (`agent` *shortname longname*) – the *shortname* (which cannot contain white-space characters) is used to refer to the agent when sending messages whereas the *longname* is used in the box at the top of the lifeline. This avoids having to repeatedly type the long agent name in messages.
- Define messages between agents (`message`, `message-sync` and `message-async`)

  It is possible to specify asynchronous and synchronous messages – the latter have closed arrowheads, the former open arrowheads. To retain compatibility with the earlier version of the textual notation we introduce two new commands: `message-sync` and `message-async`.

  Also, although not specified explicitly, a message from an agent to itself can be written and is depicted appropriately. For example, in Figure 5 after receiving the message *Request*, Bob sends a *Reminder* message to himself, followed by sending a *Response* to Alice.
- Define the start and end of boxes (`box` and `end`). The `box` command is followed by the type of box (*e.g.*, alternative, option, break). The `end` command is optionally followed by the type of the box, if the type is given then it is possible to check that `box` and `end` commands match up.
- Define the boundary between regions within a box (`next`) and guards (`guard`). The `next` command denotes the boundary between two regions, within a box. The `guard` command is followed by the text of the guard.
- Define Ref boxes (`sub`). The `sub` command is followed by the name of the protocol being referred to.
- Define continuations (`goto` and `label`). Each of the commands is followed by the name of the element.
- Define the end of an agent's participation in the protocol (`stop`)
- Define timing constraints. This is done using a command to specify the start of a timing constraint, and a second command to denote the end. Timing constraints cannot be nested, and both the start and end commands (respectively **timestart** and **timeend**) must be followed by a message.[8] For example, the following code was used to generate Figure 2:

  ```
  start sd timing
  agent a Alice
  agent b Bob
  timestart
  message a b request
  timeend {0 .. 7d}
  message b a response
  finish
  ```

- Define crossed-over messages: these are done by defining a message in two parts. The first part specifies the beginning of the message using the declaration '`mes-`' which is followed by a name, used to refer to the message, the sender, recipient, and the text on the message. The second part specifies the end of the message using the declaration '`-sage`' which is followed by a name, which matches the name given in the first declaration. For example the following text corresponds to the AUML fragment in Figure 6.

```
start sd crossed
agent i Initiator
agent p Participant
mes- label i p inform-end-of-auction
message-async i p request-payment
-sage label
finish
```

Figure 5 Depiction of messages to self

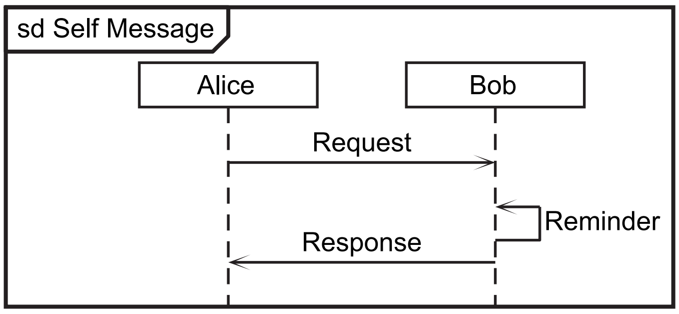


Figure 6 Example of crossed messages

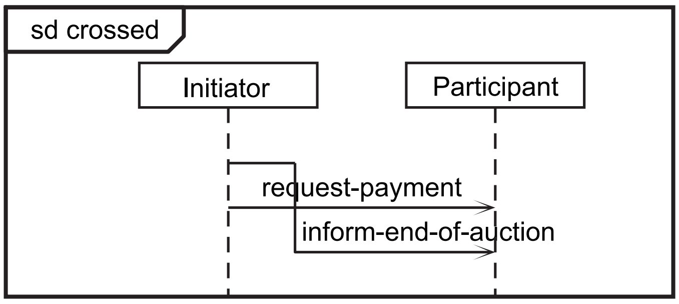


Figure 7 The AUML notation

auml ::= start agentdef* pe* **finish** ↙
start ::= **start** protocol-name ↙
agentdef ::= **agent** short-name long-name ↙
pe ::= **message** from-name to-name message-label ↙
  | **message-sync** from-name to-name message-label ↙
  | **message-async** from-name to-name message-label ↙
  | **mes-** message-name from-name to-name message-label ↙
  | **-sage** message-name ↙
  | boxstart boxcontents boxend
  | **sub** protocol-name ↙
  | **goto** label-name ↙
  | **label** label-name ↙
  | **stop** agent-name ↙
  | **timestart** ↙
  | **timeend** text ↙
boxstart ::= **box** boxtype ↙
  | **box** boxtype ↙ **guard** text ↙
boxtype ::= Alternative | Option | Break | Loop | ...
boxend ::= **end** boxtype ↙ | **end** ↙
boxcontents ::= pe* | pe* nextregion boxcontents
nextregion ::= **next** ↙ | **next** ↙ **guard** text ↙

"pe" is short for "protocol element"
Entities ending with "name" are strings, as are "text"
and "message-label".
"↙" is used to denote a newline.
Terminal symbols are in **bold** and "*" is "zero or more".

For example, the following textual protocol corresponds to Figure 4. Note that the indentation is ignored by the tool, although it is, of course, very important to human readers.

```
start sd English Auction
agent i Initiator
agent p Participant
```

```
message i p inform-start-of-auction
box loop
message i p cfp
box alternative
    message p i not-understood(syntax-error)
    stop p
next
    message p i not-understood(ontology)
    stop p
next
    message p i propose-price
    box alternative
        message i p accept
    next
        message i p reject
    end alternative
end alternative
end loop
message i p inform-end-of-auction
box option
    guard [actual-price >= reserved-price]
    message i p request
end option
finish
```

4 Tool support for AUML

In this section, we describe four different tools that use the textual notation presented in the previous section. The first tool,[9] which was the first developed, takes as input a protocol described in the textual AUML notation and produces as output an Encapsulated PostScript (EPS) file containing a rendered depiction of the protocol in the standard AUML graphical notation. In doing this the tool automatically computes layout and places the graphical elements in a visually attractive manner (*e.g.*, see Figures 1, 2, 4, 5, 6 and 8).

Although the tool lays out the interaction protocol's graphical elements automatically, and generally does a fairly good job, sometimes it is desirable to manually fine-tune the appearance or layout of a protocol. The tool supports this by adding a number of declarations to the textual notation defined in Section 3. These declarations include the following:

- `backup` moves the current vertical position up. For example the sequence `message` then `backup` then `message` will show the messages being sent at the same time. Another common sequence is `guard` then `backup` then `guard` to break a long guard over two lines.
- `agsep+` must come before agents have been defined, and is followed by a number. It increases the space between agents' lifelines by that amount.

- `agwidth+` is followed by a number, and increases the width of the box containing the labels on the agents' lifelines by that amount. It too must come before agents have been defined.
- `tagwidth+` is followed by a number. It increases the width of the tags of boxes by this amount.
- `inittagwidth+` is the same as `tagwidth+` but affects the initial tag containing the name of the protocol. It must come before the `start` command.

The protocol in Figure 4 was produced by adding the following declarations to the protocol specification given in the previous section:

```
inittagwidth+ 50

agsep+ 50
```

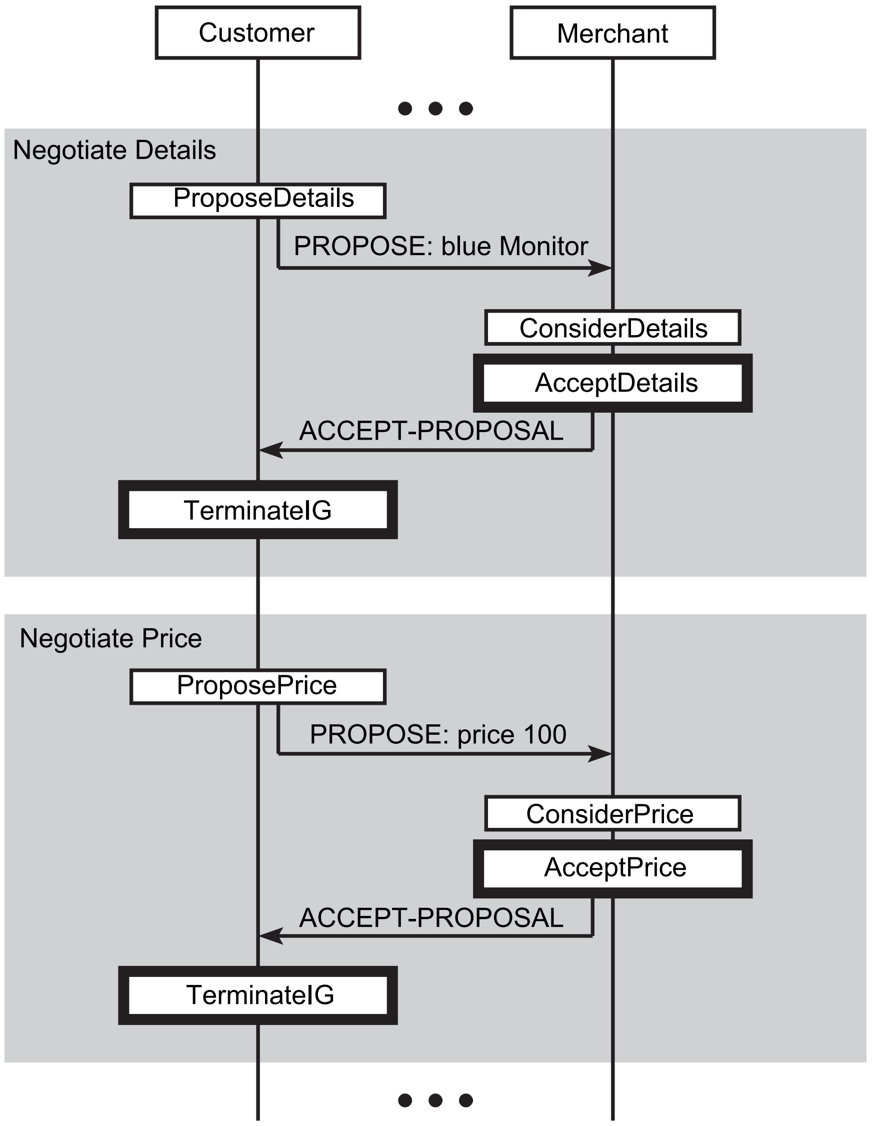
The layout algorithm used by the tool is given in the Appendix. One issue is that certain graphical elements need to be placed above other graphical elements, for example, a continuation (goto or label) should hide lifelines that are 'beneath' it. There are a number of ways of achieving this. One is to use a graphical toolkit that supports *z* coordinates. Another is to modify the algorithm so that when a continuation or sub-protocol box are drawn the lifelines are interrupted. A third option, and the one which is adopted by the tool, is to ensure that continuations and sub-protocols are drawn after lifelines are drawn. Since lifelines are drawn at the end (once the height of the protocol is known), this means that the drawing of continuations and sub-protocols needs to be *delayed.* Thus the algorithm has three phases, but only needs to read the file once:

1. Read the text file, drawing elements as they are encountered, with certain elements being placed into a queue of delayed graphical operations.
2. Once all of the text file has been read, draw the lifelines.
3. Draw all of the delayed elements in order.

Internally, the tool uses Tcl/Tk's `canvas` widget to draw the protocol and exploits its ability to export the diagram to encapsulated postscript, which can then be included in documents, or converted to a range of formats. So, in fact, the tool generates a Tcl/Tk script which is run to produce the encapsulated postscript file.

The tool has proven to be very easy to extend with additional constructs. For example, recent work on extending Prometheus to be more goal-oriented (Khallouf and Winikoff, 2005) added goals to interaction protocols (depicted as boxed text on agents' lifelines). The tool was extended with a new command `goal` *agentname goal.* Another, more significant, extension was done in the context of the *Hermes* methodology for designing flexible agent interactions (Cheong and Winikoff, 2005). One of the design artefacts produced is an *Action Message Diagram* (see Figure 8 for an example) which depicts actions (boxed text on agent lifelines), along with messages that they trigger. Actions can be final (depicted with a thicker border) and take place in order to achieve *Interaction Goals.* This is shown with a grey shaded region for each Interaction Goal. Again, the tool was extended with new commands for expressing these constructs. Both these modifications were very easy to perform: the tool was extended in a matter of a few hours by the author.

Figure 8 Action message diagram

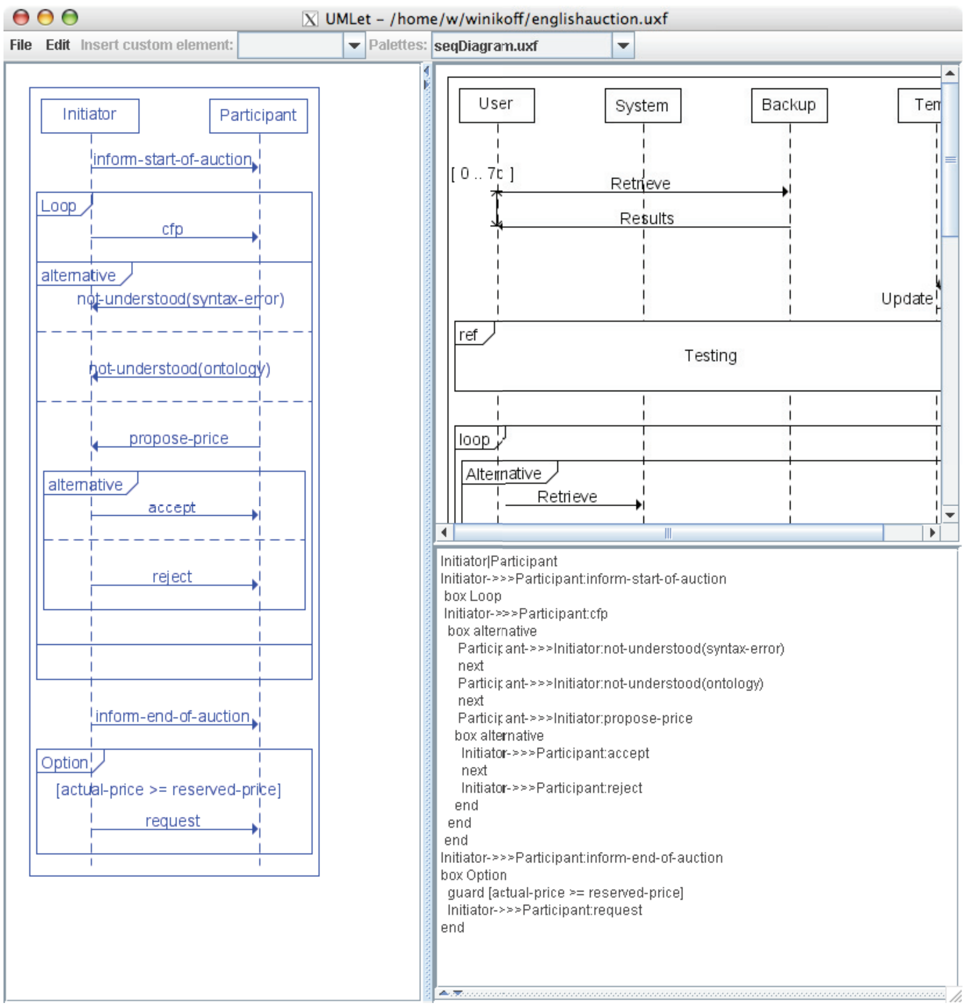


Source: From Cheong and Winikoff (2005)

The second tool that supports the textual notation is an *interactive* AUML editor. Whereas the first tool operates in batch mode, the second tool shows both the textual notation and the graphical view simultaneously, and updates the graphical view in real-time as the text is changed (see Figure 9).

This tool, which was built over a period of eight or so weeks by an undergraduate student, extends UMLet with a new diagram type. The top right hand side pane in Figure 9 is the palette, showing an example protocol that the user clones and then modifies. The English auction protocol is shown in the left hand side pane, with the textual notation view being in the bottom right hand side pane.

Figure 9 The UMLet tool



```
Initiator|Participant
Initiator->>>Participant:inform-start-of-auction
 box Loop
 Initiator->>>Participant:cfp
  box alternative
    Participant->>>Initiator:not-understood(syntax-error)
    next
    Participant->>>Initiator:not-understood(ontology)
    next
    Participant->>>Initiator:propose-price
   box alternative
    Initiator->>>Participant:accept
    next
    Initiator->>>Participant:reject
   end
  end
 end
Initiator->>>Participant:inform-end-of-auction
box Option
  guard [actual-price >= reserved-price]
  Initiator->>>Participant:request
end
```

Note that this UMLet-based tool modifies the textual notation in a few ways. Firstly, the declarations for agents follows the form used for declaring objects in UMLet interaction diagrams: they are declared initially as a list of names separated with bars, also UMLet does not use the `start` and `finish` commands. Secondly, the declaration for a message is given using the agent names (there is no distinction between short and long names), and using the syntax *from_agent arrow to_agent : message_label* where the *arrow* can be a number of different things to denote different arrow types, as is used elsewhere in UMLet. Thirdly, the syntax for specifying a sub-protocol is a little more verbose:

```
box ref
  sub name of protocol
end
```

as opposed to just `sub` *name of protocol.* Finally, the commands for depicting time constraints are different (**timestart** and **timeend** versus **begin** and **finish**) and in the UMLet-based tool the textual annotation, *i.e.*, the constraint, is given with the **begin**, rather than with the **timeend**.

The English auction example using the UMLet tool is given below:[10]

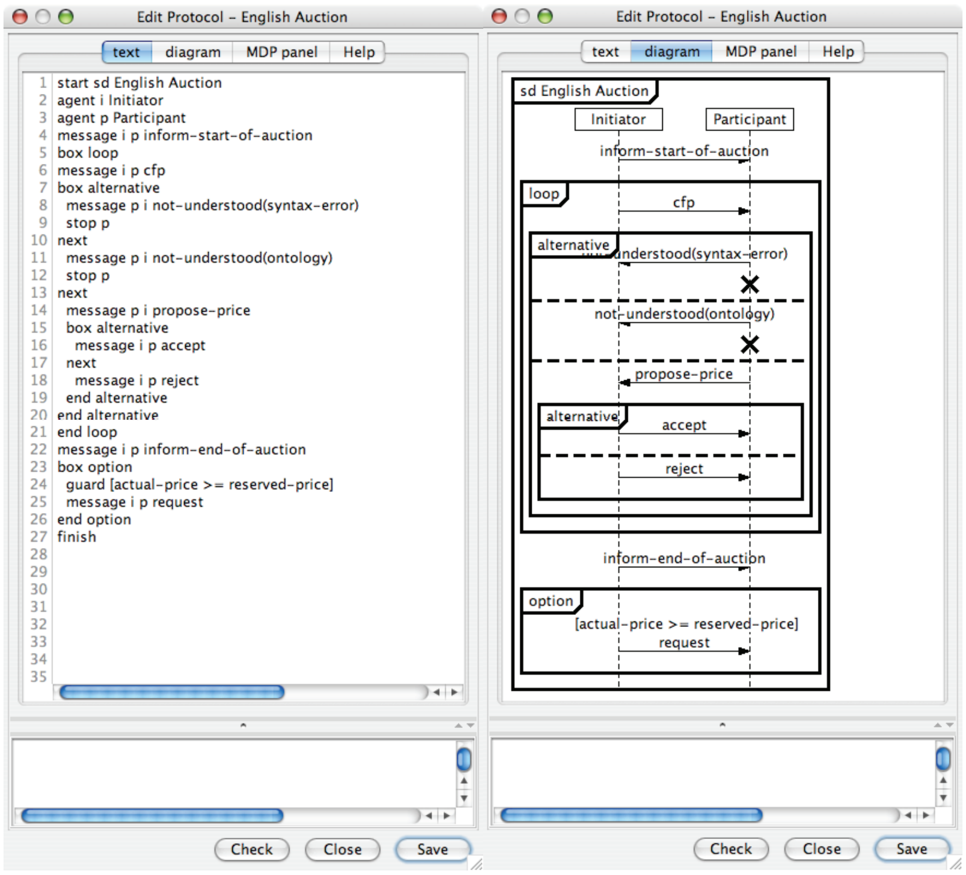
```
Initiator|Participant
Initiator->>>Participant:inform-start-of-auction
  box Loop
  Initiator->>>Participant:cfp
    box alternative
        Participant->>>Initiator:not-understood(syntax-error)
        next
        Participant->>>Initiator:not-understood(ontology)
        next
        Participant->>>Initiator:propose-price
      box alternative
          Initiator->>>Participant:accept
          next
          Initiator->>>Participant:reject
      end
    end
  end
Initiator->>>Participant:inform-end-of-auction
box Option
    guard [actual-price >= reserved-price]
    Initiator->>>Participant:request
end
```

The third tool is the Prometheus Design Tool (PDT), a freely available[11] tool supporting the design of agent systems using the Prometheus methodology. PDT has been recently extended with support for specifying protocols using the AUML textual notation described in this paper. Figure 10 shows the relevant dialog box showing the textual notation (left) and the resulting rendered protocol (right).

In addition to the commands described in Section 3, a number of additional commands are supported. These provide features which are not part of AUML, but which are useful in the context of Prometheus. Specifically, we allow life-lines corresponding to actors and to the environment to be included. Actors are depicted with dashed boxes and the environment is depicted as an actor called '{environment}' or can be simply invisible. We also allow actions and percepts to be included in the sequence diagram. An action is depicted as a message, with name enclosed in angle brackets ('<*action name*>'), from an agent to either an actor or the environment; and a percept is depicted as a message, with name enclosed in angle brackets ('>*percept name*<'), to an agent from either an actor or the environment.

The AUML protocol is integrated with the rest of PDT: when the protocol is saved entities declared in the protocol (*e.g.*, agents and messages) are propagated to the rest of the model.

Figure 10 PDT AUML support



```
start sd English Auction
agent i Initiator
agent p Participant
message i p inform-start-of-auction
box loop
message i p cfp
box alternative
  message p i not-understood(syntax-error)
  stop p
next
  message p i not-understood(ontology)
  stop p
next
  message p i propose-price
  box alternative
    message i p accept
  next
    message i p reject
  end alternative
end alternative
end loop
message i p inform-end-of-auction
box option
  guard [actual-price >= reserved-price]
  message i p request
end option
finish
```

The fourth and final tool which uses the textual notation was developed by Casella and Mascardi (2006). The primary aim of the tool is to convert AUML protocols to WS-BPEL. The tool operates in batch mode, taking as input an XMI[12] file, and producing as output a number of files, including a WSDL document, a WS-BPEL document, and the protocol in the textual notation. This demonstrates the ease of using the textual notation as an interchange format between tools. The format, unlike XML-based formats, is concise and human readable.

5 Related work

Huget (2002) has proposed an XML-based machine-readable representation for AUML protocols, called AXF. However, the notation, which is based on the earlier version of AUML, is considerably more verbose: the English auction protocol is encoded in 250 lines of AXF. Additionally, the notation is not human-readable. Although our notation is not XML-based (and hence less 'buzzword-compliant'), it is just as formal, easier to parse, and considerably easier for humans to read and write.

Similarly, although it would be possible to use a subset of XMI to represent sequence diagrams, the result is verbose, and is not easy for humans to read or write. The tool of Casella and Mascardi (2006) takes XMI as input and generates AUML, it would also be useful to be able to go in the other direction: to take the human-friendly textual notation of this paper and generate from it XMI. This is left for future work.

Doi *et al.* (2004) have proposed a textual notation for describing protocols. The notation is not intended to capture AUML protocols, but to serve as a stepping-stone between AUML and implementation. Their notation covers only a part of AUML.[13] They give an encoding for the English auction protocol which takes around 130 lines of IOM/T. However, the IOM/T description does include additional details, such as the contents of messages.

By contrast, using our notation the English Protocol is under 30 lines.

Tool support for AUML is fairly limited. Viper (Rooney *et al.*, 2004) is a graphical editor for the earlier version of AUML, which is very different to the current version. The only tool that we are aware of that supports the current version of AUML is the Ingenias Development Kit (IDK) which includes support for an AUML-like notation for protocols. However, this seems to be in alpha version and is limited ('So far, only alternatives, basic messages, and subprotocols have been implemented.' – IDK reference manual,[14] page 30). It is not clear when this limitation is likely to be removed – it was present in May 2005, and is still unchanged in version 2.5 of the IDK, described in version 2.5.2 of the manual (November 2005).

Recent work by Ehrler and Cranefield (2004) has described a tool for executing AUML protocols. Protocols are created by directly editing XML corresponding to the meta-model, *i.e.*, there is no graphical editor. Additionally, the AUML protocols are augmented with additional information, and at the time of publication of (Ehrler and Cranefield, 2004) (mid-2004) the PAUL[15] tool could only handle the Alternative box type and was limited to two agent lifelines. It appears that work on PAUL has not progressed further since 2004 (presumably due to one of the authors completing their studies and changing countries).

Although UML 2.0 is supported by tools, there are differences between UML 2.0 and AUML. For example, in UML 2.0 object lifelines have activation boxes showing when objects are active. For example, Casella and Mascardi (2006) uses the Poseidon UML tool to create AUML diagrams which are then exported to XMI format. However, the notation supported by Poseidon is, apart from the activation boxes, a subset of AUML, for instance continuations are not supported.

6 Conclusion

We have highlighted two issues that AUML suffers from – the lack of a precise formal definition, and the lack of tool support – and argued that by using a textual notation we could easily define AUML's syntax, and facilitate the task of providing tool support.

We defined a textual notation that is simple and concise, then presented evidence for the facilitation of tool support in the form of four tools that make use of the textual notation.

The notation and the first tool have been used by colleagues of the author and by students (both postgraduate and undergraduate). Our experiences have been very positive: the textual notation is very easy to learn and it provides a surprisingly practical way of capturing AUML sequence diagrams, and with the first tool, generating the standard graphical rendition. Writing interaction protocols textually is quite fast, and in particular it is faster than interacting with a graphical user interface. Additionally, the rendering tool takes care of the graphical layout, automatically producing visually attractive sequences diagrams with consistent spacing, *etc.*, thus the designer is freed to focus on the logic of the protocol, rather than its presentation.

Although the tool has proven quite useful in its current form there is, as always, scope for further work. The tool supports a significant (and increasing) subset of the AUML sequence diagram notation, but still does not support all of the notation. Some aspects that are not currently supported include cardinality annotations on messages and ‘gates’.

There is also potential for developing other forms of tool support which make use of the textual notation as an input format, output format, or interchange format. The tool of Casella and Mascardi is one example. Other possibilities include converting AUML protocols to Petri nets (*e.g.*, in order to monitor the execution of protocols (Padgham *et al.*, 2005b)). It would also be possible to extend the rendering tool into an interactive protocol execution visualisation tool by highlighting messages dynamically as they are sent and received.

We have precisely defined the *syntax* of AUML, but have not addressed its *semantics.* Defining the notation’s semantics precisely (and formally) is clearly an important area for future work. Since AUML’s sequence diagram notation is quite similar to the UML 2.0 sequence diagram notation, work in this area would be expected to build on existing work on the semantics of UML 2.0 sequence diagrams (*e.g.*, Störrle, 2003; 2004).

Finally, our experience in developing AUML sequence diagrams has highlighted some areas where the notation itself lacks expressiveness, and could be improved. One key weakness concerns protocols where there are multiple instances of a given agent type, such as auctions. AUML does not provide support to clearly indicate the parallelism associated with a broadcast message and the point at which the different threads synchronise.

Acknowledgements

I would like to acknowledge the support of Agent Oriented Software Pty. Ltd. and of the Australian Research Council (ARC) under grant LP0453486. I would also like to thank Lin Padgham and David Poutakidis for discussions of issues relating to AUML. Last but not least, I would like to thank Chun Wai (‘Steven’) Tsui who extended UMLet with AUML support, and Ian Mathieson, who implemented AUML support in PDT.

References

Bergenti, F., Gleizes, M-P. and Zambonelli, F. (Eds.) (2004) *Methodologies and Software Engineering for Agent Systems*, New York: Kluwer Academic Publishing.

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A. (2004) ‘Tropos: an agent-oriented software development methodology’, *Journal of Autonomous Agents and Multi-Agent Systems*, Vol. 8, pp.203–236.

Casella, G. and Mascardi, V. (2006) ‘From AUML to WS-BPEL’, Technical Report DISI-TR-06-01, Universita di Genova, Dipartimento di Informatica e Scienze dell'Informazione (DISI).

Cheong, C. and Winikoff, M. (2005) 'Hermes: designing goal-oriented agent interactions', *Proceedings of the 6th International Workshop on Agent-Oriented Software Engineering (AOSE-2005).*

DeLoach, S.A., Wood, M.F. and Sparkman, C.H. (2001) 'Multiagent systems engineering', *International Journal of Software Engineering and Knowledge Engineering*, Vol. 11, No. 3, pp.231–258.

Doi, T., Yoshioka, N., Tahara, Y. and Honiden, S. (2004) 'Bridging the gap between AUML and implementation using IOM/T', *Second International Workshop on Programming Multi-Agent Systems: Languages and Tools (ProMAS).*

Ehrler, L. and Cranefield, S. (2004) 'Executing agent UML diagrams', *Proceedings of the Third International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pp.906–913.

Henderson-Sellers, B. and Giorgini, P. (Eds.) (2005) *Agent-Oriented Methodologies*, Idea Group.

Huget, M-P. (2002) 'A language for exchanging agent UML protocol diagrams', Technical Report ULCS-02-009, The University of Liverpool, Computer Science Department.

Huget, M-P. and Odell, J. (2004) 'Representing agent interaction protocols with agent UML', *Proceedings of the Fifth International Workshop on Agent Oriented Software Engineering (AOSE).*

Hunt, A. and Thomas, D. (2000) *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley.

Jennings, N., Kinny, D., Wooldridge, M. and Zambonelli, F. (2004) *The Gaia Methodology*, in Bergenti *et al.* (2004), Chap. 4.

Juan, T., Pearce, A. and Sterling, L. (2002) 'ROADMAP: extending the Gaia methodology for complex open systems', *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002)*, ACM Press, pp.3–10.

Khallouf, J. and Winikoff, M. (2005) 'Towards goal-oriented design of agent systems', *First International Workshop on Integration of Software Engineering and Agent Technology (ISEAT 2005).*

OMG (2003) *UML 2.0 Superstructure Specification*, Object Management Group, document ptc/03-08-02, www.omg.org.

Padgham, L. and Winikoff, M. (2004) *Developing Intelligent Agent Systems: A Practical Guide*, John Wiley and Sons, ISBN 0-470-86120-7.

Padgham, L., Thangarajah, J. and Winikoff, M. (2005a) 'Tool support for agent development using the Prometheus methodology', *First International Workshop on Integration of Software Engineering and Agent Technology (ISEAT 2005).*

Padgham, L., Winikoff, M. and Poutakidis, D. (2005b) 'Adding debugging support to the Prometheus methodology', *Engineering Applications of Artificial Intelligence*, Special Issue on Agent-oriented Software Development, Vol. 18, No. 2, pp.173–190.

Raymond, E.S. (2004) *The Art of UNIX Programming*, Addison-Wesley.

Rooney, C., Collier, R. and O'Hare, G.M.P. (2004) 'VIPER: VIsual Protocol EditoR', *6th International Conference on Coordination Languages and Models (COORDINATION 2004).*

Spinellis, D. (2003) 'On the declarative specification of models', *IEEE Software*, doi:10.1109/MS.2003.1184181, Vol. 2, No. 20, pp.94–96.

Störrle, H. (2003) 'Semantics of interactions in UML 2.0', *Intl. Ws. Visual Languages and Formal Methods, at HCC'03*, Auckland, NZ.

Störrle, H. (2004) 'Trace semantics of UML 2.0 interactions', Technical Report TR 0403, University of Munich.

Winikoff, M. (2005) 'Towards making agent UML practical: a textual notation and a tool', *First International Workshop on Integration of Software Engineering and Agent Technology (ISEAT 2005).*

Notes

1 http://www.uml.org

2 http://www.auml.org

3 The Foundation for Intelligent Physical Agents, http://www.fipa.org.

4 http://www.umlet.com

5 http://www.cs.rmit.edu.au/agents/pdt/

6 Actually the label of a lifeline can contain a type and/or an instance name, as well as other information.

7 Actually it is possible for none of the regions to be executed if all guards are false. This can be avoided by having an 'else' guard.

8 Neither of these constraints are captured by the grammar in Figure 7.

9 The tool is freely available from http://www.winikoff.net/auml.

10 At the time of writing the tool did not support the 'stop' command.

11 http://www.cs.rmit.edu.au/agents/pdt/

12 XML Metadata Interchange, http://www.omg.org/technology/documents/formal/xmi.htm.

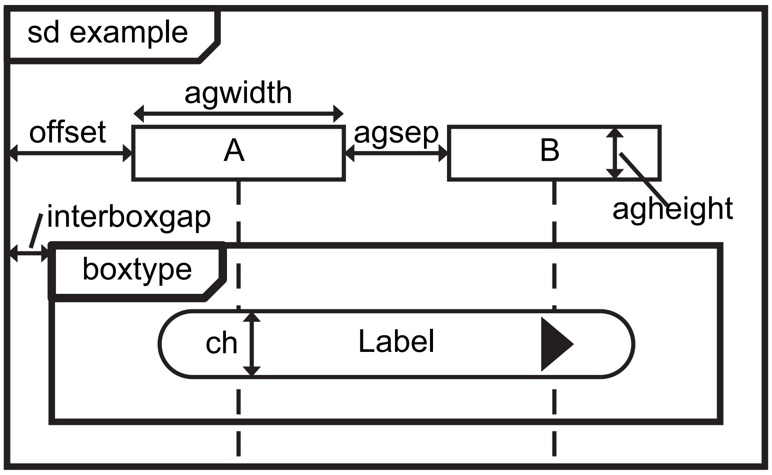
13 "The current version [sic] IOM/T can not fully represent interactions which are equivalent to design in AUML … there are not the notations which represent CombinedFragment. Only CombinedFragments whose interaction operator [sic] are 'Loop' can be represented by 'while' structure" (Doi *et al.*, 2004, Section 2.7).

14 http://ingenias.sourceforge.net (accessed 7 March 2006).

15 **P**lug-in for **A**gent **U**ML **L**inking.

Appendix The layout algorithm

The algorithm below uses a number of constants, corresponding to lengths, which are illustrated below.



```
agentsx ← {} //map agent names to x coordinates
agentsy ← {} //maps agent names to y coordinates
boxes ← {} //stack of entries, each of the form (box_type,y)
crossed ← {} //maps message names to details
agentnum ← 0
vp ← 0 //vertical position (i.e. y)
time_y ← 0 //y coordinate of timestart
while more input do
  line ← read_line()
  (type,line) ← get_first_word(line) //type is the first word, line is the remaining text
  if type = "agent" then
    (shortname,longname) ← get_first_word(line)
    agentsy[shortname] ← vp
    x ← agentnum × (agwidth + agsep) + offset
    agentsx[shortname] ← x + agwidth/2
    draw_rectangle(x, vp, x + agwidth, vp + agheight)
    draw_centered_text(agentsx[shortname], vp, longname)
    agentnum ← agentnum + 1
    if next line is not an agent declaration then
      increase vp
  else if type ∈ {message, message-sync, message-async} then
    (from,line) ← get_first_word(line)
    (to,text) ← get_first_word(line)
    x_from ← agentsx[from]
    x_to ← agentsx[to]
    if x_from = x_to then
      draw_self_message(x_from,vp,text)
    else
      draw_arrow(x_from,vp,x_to,vp) //Draw appropriate arrow for “message-sync”
      and “messsage-async”
      draw_centered_text((x_from + x_to)/2,vp,text)
    increase vp
  else if type = “box” OR type = “start” then
    boxes ← push(boxes,⟨line, vp⟩)
    indent ← interboxgap × size(boxes)
    draw_tag(indent, vp,line) //Draw $\boxed{\text{boxtype}}$
    increase vp
  else if type = “next” then
    indent ← interboxgap × size(boxes)
    x ← offset + agentnum × (agwidth + agsep) – indent
    draw_heavy_dashed_line(indent, vp, x, vp)
    increase vp
  else if type = “end” or type = “finish” then
    indent ← interboxgap × size(boxes)
    x ← offset + agentnum × (agwidth + agsep) – indent
    ⟨boxtype, y⟩ ← pop(boxes) //pop from top of stack
    if type = “end” and boxtype ≠ line then
      print “Error: closing box type” + line + “doesn’t match opening box type” + boxtype
    else
      draw_rectangle(indent, y, x, vp)
      if type = “end” then
        increase vp
  else if type = “guard” then
    x ← agentsx[0]
    delayed_draw_filled_white_rectangle(x–2,vp–5,x+2,vp) //Erase covered lifeline
    delayed_draw_text(x,vp,line)
    increase vp
  else if type = “goto” or type = “label” then
    x1 ← agentsx[0]
    x2 ← agentsx[agentnum-1] //last agent’s x coordinate
    y ← vp + ch //ch is the height of the continuation
    delayed_draw_filled_roundrect(x1 – ch,vp, x2 + ch,y)
    delayed_draw_centered_text((x1 + x2)/2, vp + (ch/2),line)
    if type = “goto” then
      x1 ← x2 //If goto then the triangle is at the right side
    delayed_draw_triangle(x1-tr,vp+2,x1+tr,vp+ch/2,x1-tr,y-2)
    //Draw ►, tr is half the triangle’s width
    increase vp
  else if type = “stop” then
    draw_X(agentsx[line],vp)
    increase vp
  else if type = “sub” then
    indent ← interboxgap × (1 + size(boxes)) //“+1” because we don’t push sub onto boxes
    x ← offset + agentnum × (agwidth + agsep) – indent
```

```
        draw_tag(indent, vp, “ref”)
        oldvp = vp
        increase vp
        delayed_draw_filled_rectangle(indent, oldvp, x, vp)
        delayed_draw_centered_text( (indent+x)/2,(oldvp+vp)/2, line)
    **else if** type = “timestart” **then**
        time_y = vp
    **else if** type = “timeend” **then**
        draw_doubleheadedarrow(agentsx[0]-to,time_y,agentsx[0]-to,vp)
        // “to” is the offset to the left
        draw_line(agentsx[0]-(2 × to),time_y,agentsx[0],time_y)
        draw_line(agentsx[0]-(2 × to),vp,agentsx[0],vp)
        draw_right_text(agentsx[0] – (1.5 × to), (vp + time_y) /2, line)
        // “right” = right aligned text
        //Note that we do not increase vp
    **else if** type = “mes-” **then**
        (message-name,line) ← get_first_word(line)
        (from,line) ← get_first_word(line)
        (to,line) ← get_first_word(line)
        crossed[message-name] ← ⟨vp, from, to, line⟩
        crosspoint ← (agentsx[from] × 0.75) + (agentsx[to] × 0.25)
        draw_line(agentsx[from],vp,crosspoint,vp)
        increase vp
    **else** if type = “-sage” **then**
        (name,line) ← get_first_word(line)
        **if not defined** crossed[message-name] **then**
            **print** “Error: no crossed message named” + name
        ⟨y, from, to, line⟩ ← crossed[message-name]
        crosspoint ← (agentsx[from] × 0.75) + (agentsx[to] × 0.25)
        textpoint ← (agentsx[from] × 0.25) + (agentsx[to] × 0.75)
        draw_line(crosspoint,y,crosspoint,vp)
        draw_arrow(crosspoint,vp,agentsx[to],vp)
        draw_centered_text (textpoint,vp,text)
        increase vp
**for** i = 0 to agentnum – 1 **do**
    draw_dashed_line(agentsx[i],agentsy [i]+agheight,agentsx[i],vp) //Draw lifelines
    Draw delayed items
```

Note that this algorithm does not draw ‘bridges’ over crossed messages. This can be done by extending the arrow drawing in messages to check for outstanding crossed messages, breaking the arrow into multiple lines (the last of which is an arrow), and adding ‘bridges’. This extension is straight-forward, but the details are somewhat verbose, and so it is not shown here.