Agent-K: An Integration of AOP and KQML

Winton H E Davies & Peter Edwards

Department of Computing Science
King's College
University of Aberdeen
Aberdeen, AB9 2UE, UK.

Email: {wdavies, pedwards}@csd.abdn.ac.uk

1 Introduction

This paper describes a synthesis of two well-known agent paradigms: Agent-Oriented Programming, Shoham (1990), and the Knowledge Query & Manipulation Language, Finin (1993). The initial implementation of AOP, Agent-0, is a simple language for specifying agent behaviour. KQML provides a standard language for inter-agent communication. Our integration (which we have called Agent-K) demonstrates that Agent-0 and KQML are highly compatible. Agent-K provides the possibility of inter-operable (or open) software agents, that can communicate via KQML and which are programmed using the AOP approach.

We begin with an overview of AOP and KQML before going on to describe our motivations for this work. This is followed by a description of the design and implementation of Agent-K[1]. We conclude with a discussion of the issues raised by the integration of AOP and KQML.

1.1 Agent-Oriented Programming (AOP)

AOP is a specialization of Object-Oriented Programming (OOP). Agents are objects that have mental states (such as beliefs, desires and intentions) and a notion of time. They are programmed using *commitment rules* . These are simple forward chaining rules which connect messages, the agent's internal state and its actions.

Agent-0 is an implementation of the AOP principle which supports three fundamental mental states: Belief, Capability and Commitment. Beliefs are logical statements about the world that the agent believes to be true or false. Capabilities are actions the agent is able to perform. Commitments are guarantees that an agent will carry out an action at a certain time. Three basic message types are provided: *inform, request* and *unrequest*. *Inform* allows agents to communicate mental states to other agents, whilst *request* allows an agent to ask another agent to perform an action, and *unrequest* cancels a *request*. Commitment rules consist of antecedent conditions that are matched against incoming messages and the agent's internal states (such as beliefs). If a rule is triggered, the agent will generate a commitment to: *do* a private action, *refrain* from an action, or send a message (i.e. *inform,* etc.). These commitments are then executed at the time specified by the agent who sent the request.

[1] Agent-K is available by anonymous ftp from: *ftp.csd.abdn.ac.uk:/pub/wdavies/agentk*

PLACA, Thomas (1993), is an extension of the AOP concept. It has expanded the underlying logic of mental states to include *intentions* (a commitment to achieve a state of the world). In parallel with this, actions may be planned. This means that multiple actions (private actions, message sending, etc.) are composed into plans, which are compared with the agent's environment. Once the agent has determined a suitable plan, it will make a commitment to execute it. This contrasts with Agent-0, where a single commitment is generated for each successful rule match.

1.2 The Knowledge Query & Manipulation Language (KQML)

KQML is part of the Knowledge Sharing Effort (KSE), Patil (1992). The KSE is a series of initiatives to provide re-usable and open knowledge-bases. KQML provides a standard language for communication to and between knowledge-based systems. Other components of the KSE include the Knowledge Interchange Format (KIF) and Ontolingua, Patil (1992). KIF is a standardised knowledge representation language, whilst Ontolingua is a language for specifying standard ontologies.

KQML is based on Speech-Act Theory[2] - i.e. a message is a performative indicating what the receiver is expected to do with the message. For example, a simple "*tell*" message indicates that the receiver is expected to believe the fact(s) provided. An "*ask*" message expects an answer to a question.

KQML provides an extended list of performatives, which deal with belief revision, querying, knowledge-base maintenance, actions and services. The syntax is that of a performative followed by an unordered list of keyword-value pairs. For example:

(*ask-one* :receiver *weather-station* :sender *forecaster*
:content *rain(today, X)* :language *prolog* :reply-with *day10*)

may elicit the response:

(*reply* :receiver *forecaster* :sender *weather-station*
:content *rain(today, no)* :language *prolog* :in-reply-to *day10*)

1.3 Motivation

Our reasons behind the integration of AOP and KQML are twofold. Firstly as part of an evaluation of Agent-0 and KQML; and secondly in order to provide agents that will support our research objectives in the area of distributed data-gathering and discovery, Davies (*submitted*).

2 Design

The first issue which had to be addressed when Agent-0 and KQML were combined was how agents should handle KQML messages. The second issue was the form of the architecture. These are described below.

2.1 The Integration of KQML and AOP

There are at least four ways in which an AOP agent could handle KQML messages. These are summarised in Fig 1. The main difference between the Agent-0 and KQML message languages is the number of message types. Agent-0 has

---

[2] The Agent-0 messages *request, inform and unrequest* also derive from Speech-Act theory.

three, whilst KQML currently supplies over twenty. However, Agent-0 is able to match the capabilities of KQML by nesting messages. For example, to ask a question, a *request(inform(?x))* message is sent. KQML can also nest messages, but does not need to do this for simple tasks such as asking a question.

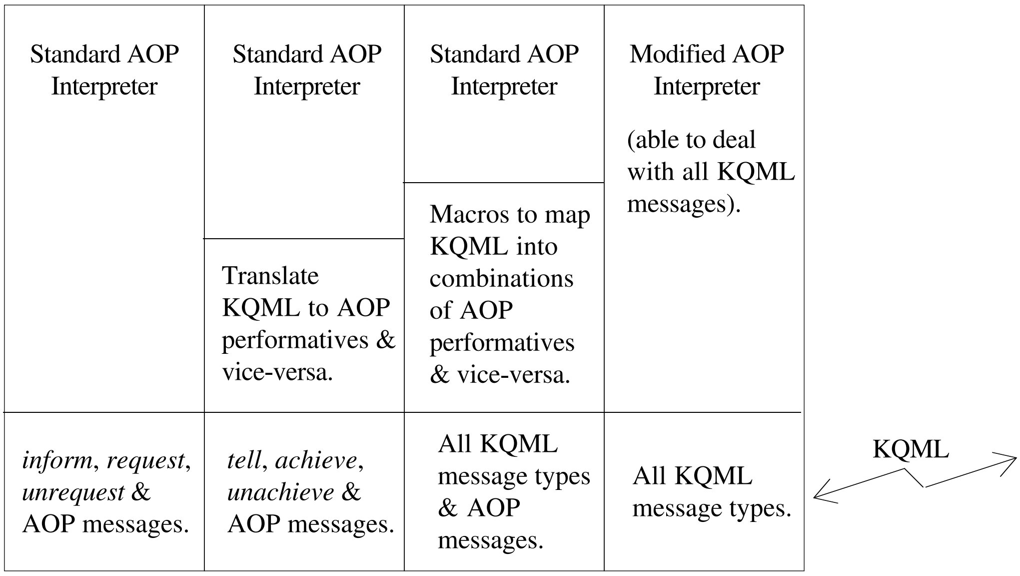

Figure 1: Possible Approaches to Integration

The first option is to extend KQML to include *inform, request* and *unrequest*. This was felt to be unrealistic, as it would require other KQML agents to account for the nature of Agent-K agents. The second method is to perform a one-to-one mapping between similar messages, i.e. between the Agent-0 messages *inform, request* and *unrequest* and the KQML equivalents *tell, achieve* and *unachieve*. This approach suffers from the same fundamental weakness as the first, i.e. it restricts our agents to communication with similar agents, rather than all agents employing KQML.

The third approach is to perform a mapping between the KQML messages and AOP messages. For example, using a macro to convert the KQML message "*ask-if*" into *inform (request (?x))*, and vice-versa for out-going messages. This is superficially attractive because it would allow AOP agents to communicate using KQML to other agents, as well as allowing communication with agents that might use the Agent-0 message language. However, this approach would be more complex to implement than the fourth approach, and appears unjustified as there is no widespread use of the Agent-0 message protocol. As KQML messages can easily replace the Agent-0 messages in commitment rules, the fourth approach was adopted. This method requires the modification of the Agent-0 interpreter to allow commitment rules to refer to KQML messages in the antecedent conditions and as consequent actions.

2.2 The Architecture of Agent-K

The Agent-K architecture (Fig. 2) consists of four components, and was constructed by combining a Prolog version of the Agent-0 interpreter[3] and a third party program library, KAPI[4] that performs the actual delivery of KQML messages. The starting point is an Agent-K program, specified in terms of commitment rules. This is loaded into the interpreter,

---

[3] David Galles, Stanford University.

[4] Jay Weber, EIT. Available by anonymous ftp from: *ftp.eit.com:/pub/shade/kapi/**

which processes the commitment rules in response to the incoming messages, the current mental state and capabilities of the agent, and the current clock time. The KAPI interface transfers KQML messages to and from other agents. This is interfaced to the interpreter by a layer of converters.

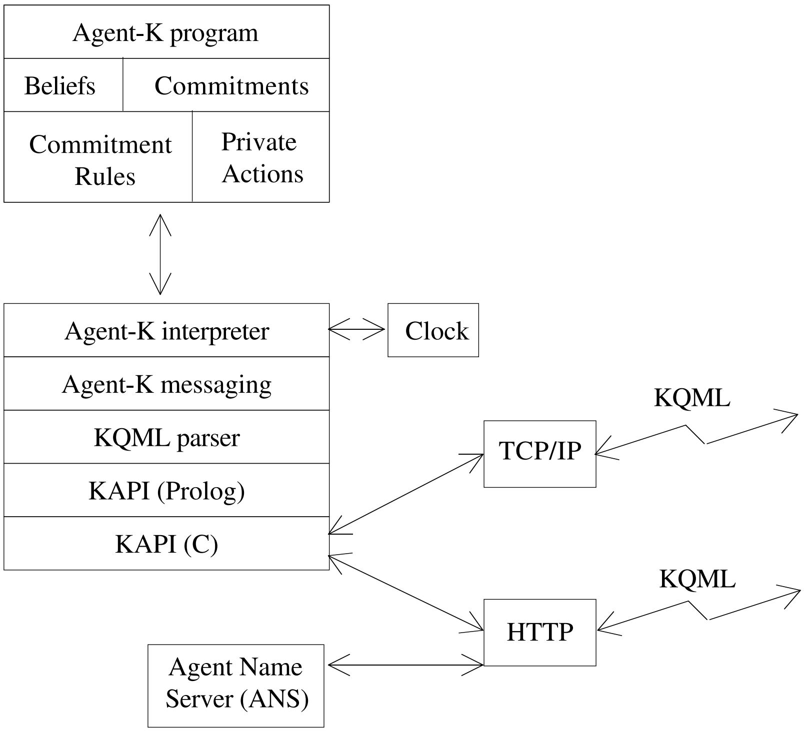

Figure 2: The Agent-K Architecture

The remaining changes to the interpreter and the commitment rules will be described in the next section.

3 Implementation

To produce Agent-K, a number of minor changes to the KAPI code and a restriction of the KQML expression syntax were necessary. Agent-0 required more extensive modifications. In this section we briefly describe these changes and include an example commitment rule and trace.

3.1 Changes to KAPI and KQML

KAPI is a library of C routines that allows a user to send KQML message strings via TCP/IP, HTTP or electronic mail. It parses the name from the :*receiver* field, and delivers the KQML message to the port specified by the Universal Resource Locator (URL) associated with the agent name. Thus we had to integrate these C routines as Prolog foreign functions. A slight problem existed in that KAPI has altered the KQML specification of the sender name, in order to allow efficient use of network channels. The most significant change we had to make was the addition of a new function (KPrologBuffer) to the KAPI module KaConnect. This function allocates a message buffer required by the function KGetString. Normally the calling program should allocate this message buffer, but Prolog has no mechanism to do this.

Two further operations were required to allow KQML messages be transferred to the interpreter level. The first was to parse the raw ASCII string representing the KQML message into tokens. The second was to convert the format of a

KQML message from that of a LISP function definition to an unordered Prolog list of unary predicates. For example:

"(tell :receiver john...)"

would become:

[tell, receiver(john),...].

A Definite Clause Grammar was employed, which parses the string for tokens and builds the list. For simplicity, it was decided not to support the full KQML expression syntax; we therefore restrict KQML expressions to being valid Prolog atoms. The resulting predicate, *kqml_prolog(KQML, PROLOG)*, converts KQML strings into Prolog, and vice-versa

3.2 From Agent-0 to Agent-K

There are four main differences between Agent-0 and Agent-K. These are:

- *Inform, request* and *unrequest* message actions have been removed. They are replaced by a single message action *kqml(time, message). Message* is a KQML message of the form *[performative, keyword(value)].* For example:

  [tell, sender(r2d2), receiver(c3po), content(good(yoda)), language(prolog)].

- Commitment rules have been changed. The message condition part is now a template that will match a required message. It can be the empty list, *[]*, which will match all messages, or of the form *[tell, content(X)]*, etc. A special case is *[nil]*, which tests the mental conditions, whether or not there are incoming messages. The mental condition component now allows general Prolog predicates (rather than just beliefs) to be used (this was specified by Shoham (1990), but not implemented in the version of Agent-0 used).

  In addition, as mentioned above, all the message actions have been replaced, and a new action *believe([Time, Fact])* has been added. This removes the inconsistency of *inform* behaviour in Agent-0 (whereby *inform* cannot be processed by commitment rules, and an agent simply has to believe whatever it is told). These changes do not appear to have substantially altered the nature of commitment rules, but arguably do make them more readable. Fig. 3 shows an example comparing an Agent-0 rule for replying to a general query, with the equivalent in Agent-K.

```
Agent-0:  commit([S, request, inform(b(T, F)],
                 [b([T, F])],
                 S,
                 INFORM(T, S, b(T, F))]).


Agent-K:  commit(['ask-one', content(b([T, F]))],
                 [clock(Now),b([T, F])],
                 S,
                 kqml(Now,[reply, receiver(S), sender(c3po),
                 content(b([T, F])), language(prolog)])).
```

Figure 3: An Example Commitment Rule

- The interpreter has been modified to use the new messaging interface. Every message received is tested against every commitment rule, in order to allow multiple actions to occur. In addition, during every cycle, the [nil] commitment rules are tested (see above).

- Each agent is now a self-contained process, consisting of a copy of the interpreter and a single agent program. The interpreter has a number of commands to help manage the agent, from loading to debugging key predicates. There is also detailed logging that illustrates commitment rule behaviour with respect to messages.

Fig. 4 shows an edited log of agent *c3po* reacting to an incoming message from agent *r2d2*. It receives the KQML message via TCP/IP, converts it, and tests each rule against the message. It matches rule 8, and makes a commitment to immediately send the answer *(good('Han Solo'))* . The interpreter then checks to see if there are any rules that may be activated simply by changes in the mental state. There are none that match, so the agent carries out its commitment to send the message via TCP/IP back to agent *r2d2*. The lists of numbers which appear in the log represent time. For example, [8,26,94,17,58,27] represents the date 26th Aug 1994 17:58:27.

```
TCPrecv receiving: (ask-all :receiver c3po :sender r2d2 :language prolog
                            :content b([[8,26,94,17,58,27],good(A)]).)

['ask-all', receiver(c3po),sender(r2d2),content(b([[8,26,94,17,58,27],good(A)])),
            language(prolog)].

1.   [[nil],[clock(A),b([A, not(alive(c3po))])]]. => do.
...
7.   [[nil],[clock(A),b([A, alive(r2d2)]),b([A, not(wait_reply)])]]. => believe.

8.   [['ask-all', content(b([A,B]))],[clock(C),b([A,B])]]. => kqml.

       => kqml([8,26,94,17,58,31], [reply, receiver(r2d2),sender(c3po),
               content(b([[8,26,94,17,58,27],good('Han Solo')])),
               language(prolog)]).

9.   [[reply, content(b([A,B]))],[]]. => believe.

10.  [[reply, content(b([A,B]))],[clock(C)]]. => do.

[nil].
1.   [[nil],[clock(A),b([A, not(alive(c3po))])]]. => do.
 ...
10.  [[reply, content(b([A,B]))],[clock(C)]]. => do.

TCPsend sending: (reply :receiver r2d2 :sender c3po :language prolog
                        :content b([[8,26,94,17,58,27],good('Han Solo')]).)
```

Figure 4. An Example Agent-K Log

4 Discussion & Future Work

The first part of this section deals with issues arising from the integration of AOP and KQML, the second deals with possible future work. As stated in the introduction, we believe that we now have a prototype system for building open software agents. However, there are many improvements that could be made.

4.1 AOP/Agent-0 Issues

The programming of Agent-0 is achieved through commitment rules which relate messages and mental conditions. These are similar to methods employed in object-oriented programming languages, Masini (1991). Given this, it may be more appropriate to replace the rules with a parameterised method for each message type. This is related to the more general question of what methodology to use for programming agents. Commitment rule programming is very similar to programming forward-chaining production rule systems. As a result, programming Agent-0 has the same benefits and drawbacks as programming in production rule languages such as OPS5, Brownston (1985), i.e. flow of control and program structure can be unclear. This leads to problems in encoding complex behaviours, e.g. distributed problem solving or processing of lists of beliefs.

It would appear that the nesting of messages in AOP (e.g. *request(inform(?x))*) is unnecessary, as the extended KQML list appears to achieve the same effects, without nesting. Finally, integrating Agent-0 and KQML has highlighted the difference between refraining and decommiting from an action. In Agent-0 the *refrain* message implies that an action should never be performed, whereas *unrequest* cancels a previous *request* for an action. We would like to see KQML reflect the difference between refraining and decommiting; currently it simply has the *unachieve* message. Furthermore, *unachieve* could also be interpreted as a request to undo an action (a behaviour which Agent-0 does not support).

4.2 KQML Issues

KQML appears to be well specified. However, differences between 'informative' performatives (such as *tell*) and 'database' performatives (such as *insert*) raise a number of issues, as do the exact semantics of *achieve* and *unachieve*. A second issue is that whilst there is a *:language* keyword for the *:content* field, a *:keyword-language* keyword should be given for the remaining fields such as *:sender*, etc. This would make it simpler for non-LISP systems employing KQML to use these fields. However, this then raises the question of how an agent could read the *:keyword-language* value in the first place. A final plea would be for KQML messages to carry a time field.

4.3 Agent-K Issues

Agent-K agents are more practical than Agent-0 agents, as they are now able to communicate across a network, both with other Agent-K agents, and other KQML-based systems. However, they are far from perfect. Agent-K has many of the same weaknesses as Agent-0. In addition, the use of Prolog for beliefs and commitments restricts Agent-K agents to interaction with other Prolog-based agents. While using KQML means that our agents can communicate with other agents, this does not mean that they can understand the *:content* field. This should be addressed by using KIF either as an intermediary language, or as the internal agent language.

There are some minor drawbacks to Agent-K. Firstly, there is no way to handle message delivery failures. This is because commitment rules do not provide a mechanism for this. However, this could be solved by storing failure messages as beliefs. Secondly, the storage of beliefs about commitments is inefficient; an agent stores a belief about the outcome of every commitment rule that is fired. Thirdly, there should be a way to state initial agent goals without encoding them as initialisation commitment rules.

4.4 Future Work

Our first goal is to apply Agent-K to a series of increasingly more complex tasks. These are a distributed problem solving testbed, a multi-agent learning system, and finally a distributed data- mining system.

The second is to augment Agent-K. There are a number of internal improvements that can be made, e.g. error handling. We also hope to incorporate some of the ideas contained within the PLACA system (such as planning and intentions). Finally, we plan to construct a simple user interface agent which can instruct Agent-K agents using KQML.

5 Acknowledgements

The work described here is supported by the UK Engineering and Physical Science Research Council (EPSRC). We wish to acknowledge the assistance of Yoav Shoham, Tim Finin, Jay Weber, Becky Thomas and David Galles for various helpful comments and for providing the original Agent-0 and KAPI code. Thanks also to Ciara Byrne and Bob Holton for comments on an earlier draft of this document.

6 Bibliography

Brownston (1985) L. Brownston et. al., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming,* Addison -Wesley, 1985.

Davies (submitted) W. Davies & P. Edwards, *Agent-Based Knowledge-Discovery*, submitted to the AAAI-95 Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments.

Finin (1993) T. Finin et. al., *DRAFT Specification of the KQML Agent-Communication Language*, unpublished draft, 1993.

Masini (1991) G. Masini et. al., *Object-Oriented Languages*, A.P.I.C. Series No. 34, Academic Press, New York, 1991.

Patil (1992) R.S. Patil, R.E. Fikes, P.F. Patel-Schneider, D. MacKay, T. Finin, T. Gruber, & R. Neches, *The DARPA Knowledge Sharing Effort: Progress Report,* In Proceedings of KR'92 - The Annual International Conference on Knowledge Representation, Cambridge, MA, 1992.

Shoham (1990) Y. Shoham, *Agent-Oriented Programming*, Technical Report No. STAN-CS-90-1335, Computer Science Department, Stanford University, 1990.

Thomas (1993) S.R. Thomas, *PLACA, an Agent Oriented Programming Language*, Ph.D. Dissertation, Computer Science Department, Stanford University, 1993.