

AUCS/TR9406

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,
Scotland, UK
AB9 2UE.

Abstract

This report 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 interoperable (or open) software agents, that can communicate via KQML and which are programmed using the AOP approach.

Contents

1	Introduction	1
1.1	Agent-Oriented Programming (AOP)	1
1.2	The Knowledge Query & Manipulation Language (KQML)	1
1.3	Motivation	2
2	Design	2
2.1	The Integration of KQML and AOP	2
2.2	The Architecture of Agent-K	3
3	Implementation	4
3.1	Modifications to KAPI and KQML	4
3.2	From Agent-0 to Agent-K	5
3.3	The Agent-K Syntax	6
4	Programming Agent-K	7
4.1	An Agent Program	7
4.2	Commitment Rule Execution	9
4.3	Agent Interaction	10
5	Discussion & Future Work	11
5.1	AOP/Agent-0 Issues	11
5.2	KQML Issues	12
5.3	Agent-K Issues	13
5.4	Future Work	14
6	Acknowledgements	14
7	Bibliography	14

1 Introduction

This report 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 interoperable (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-K¹. 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.

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 based on the agent's beliefs and capabilities. 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.

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

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 theory², Searle (1969), 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 (1995).

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 exact form of the integration model. 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 Figure 1. The main difference between the Agent-0 and KQML message languages is the number of message types. Agent-0 has 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.

²The Agent-0 messages *request*, *inform* and *unrequest* also derive from speech-act theory.

Standard AOP Interpreter	Standard AOP Interpreter	Standard AOP Interpreter	Modified AOP Interpreter (able to deal with all KQML messages).
	Translate KQML to AOP performatives & vice-versa.	Macros to map KQML into combinations of AOP performatives & vice-versa.	
<i>inform, request, unrequest</i> & AOP messages.	<i>tell, achieve, unachieve</i> & AOP messages.	All KQML message types & AOP messages.	All KQML message types.




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 Agent-K Integration Model

The Agent-K integration model (Figure 2) consists of four components, and was constructed by combining a Prolog version of the Agent-0 interpreter³ and a third party program library, KAPI⁴ that performs the actual delivery of KQML messages. The starting point is an Agent-K program, specified in terms of

³ David Galle, Stanford University.

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

commitment rules. This is loaded into the interpreter, 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.

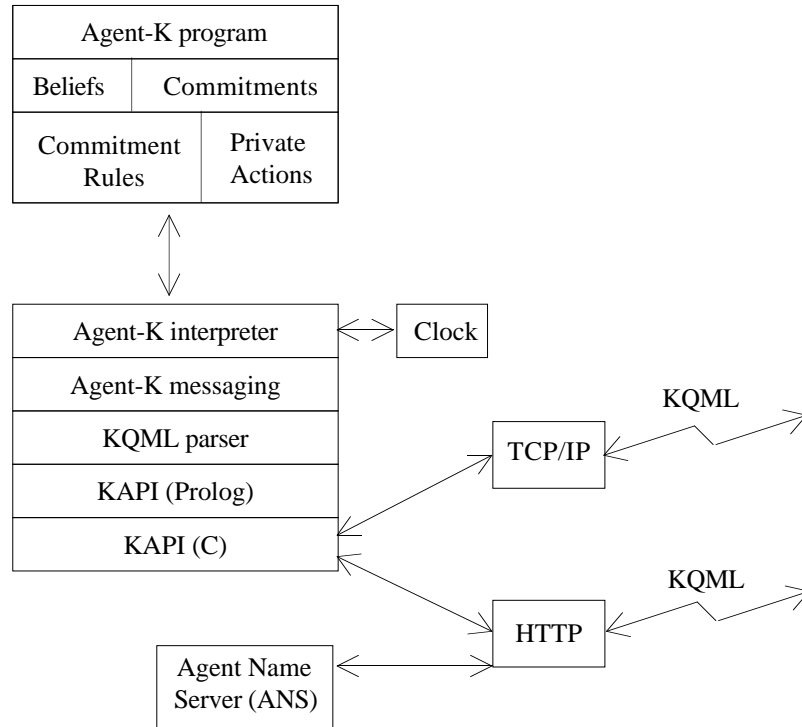


Figure 2: The Agent-K Integration Model

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 the full BNF for the Agent-K language. Following this, in section 4, we provide a simple scenario that demonstrates the use of Agent-K.

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 we used as the basis for Agent-K).

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.

- 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.

Figure 3 compares an Agent-0 commitment rule with the equivalent in Agent-K. Both rules reply to a general query. The message condition of the Agent-0 rule matches a request that contains a nested inform message. The Agent-K rule matches a message containing an *ask-one* performative. The *content* field is equivalent to the parameter in the inform message, i.e. $b(T, F)$, where $b(_, _)$ is a belief, T is a time, and F is a fact. The mental condition to match in both cases, is whether fact F is believed to be true at time T . In the case of Agent-K, the Prolog predicate `clock(Now)` is used to provide the current time (instantiated as the variable *Now*). In Agent-0, this is assumed to be available as a special variable. The third component of both rules specifies the agent to whom the commitment is made. The final component is the action. In the case of Agent-0, it is to perform an *inform* action. In the case of Agent-K, it is to perform a *kqml* action, which sends a reply message.

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

3.3 The Agent-K Syntax

Figure 4 gives the BNF definition of Agent-K. An Agent-K program consists of the agent's name, followed by a list of its capabilities, a list of the commitment rules it will use, followed by the definitions of any private actions. This is effectively the same as the Agent-0 syntax. The detailed differences with Agent-0 have been described in the previous section. These are the form of the messages (*Msgpatterns* and *Message*), the mental conditions (the addition of a Prolog clause as a valid element), and the addition of the *believe* and *kqml* actions. In the next section we will demonstrate the use of Agent-K.

Agent Program	
who_am_i(<Agent>)	
{can(<Agent>, do(<Time>, <Private Action>))} *	
{commit(<Msgpattern>, <Mntlcond>, <Agent>, <Action>)} *	
{<Private Action>} *	
Non-terminals	
<Agent>	::= Name of an agent
<Time>	::= <Variable> [Mon, Day, Yr, Hr, Min, Sec]
<Variable>	::= Prolog Variable
<Private Action>	::= Prolog predicate add_friend(<Agent>, <URL>) initialise_agent(<URL>)
<Msgpattern>	::= [<Message>] [nil] []
<Message>	::= [{<Performative>}{, <Keyword>(<Value>)}*]
<Performative>	::= <Variable> KQML Performative
<Keyword>	::= KQML Keyword
<Value>	::= <Variable> KQML Expression
<Mntlcond>	::= [<Mntlpattern> {, <Mntlpattern>}*]
<Mntlpattern>	::= b(<Fact>) not(Mntlcond) <i>Prolog predicate</i>
<Fact>	::= [<time>, <BeliefKernel>]
<BeliefKernel>	::= <i>Prolog Predicate</i> <Internal Belief> <Internal Commitment> <Internal Capability>
<Internal Belief>	::= b(<Agent>, <Fact>)
<Internal Commitment>	::= cmt(<Agent>, <Agent>, <Action>)
<Internal Capability>	::= can(<Agent>, <Action>)
<Action>	::= do(<Time>, <Private Action>) <i>believe</i> (<Time>, <Agent>, <Fact>) <i>kqml</i> (<Time>, <Message>) if(<Mntlcond>, <Action>)

Figure 4: BNF Agent-K Syntax

4 Programming Agent-K

In this section we provide a simple example of programming using Agent-K, and describe the resulting behaviour. The scenario is very simple and involves two agents, *C3PO* and *R2D2*, which know that Han Solo and Princess Leia are good respectively. Both agents have the same goal (encoded as a commitment rule), i.e. to determine who the other agent believes to be good.

4.1 An Agent Program

Figure 5 shows the program for agent *R2D2*. Agent *C3PO* is identical apart from its belief that Han Solo is good.

```
/* R2D2 */
who_am_i(r2d2).
say(X) :- writeln(['BEEP BEEP! > ', X]).

/* INITIALISATION */
commit([nil],
  [clock(Now), b([Now, not(alive(r2d2))])],
  nil,
  do(Now, initialise_agent('tcp://condor:9001'))).

commit([nil],
  [clock(Now), b([Now, not(alive(r2d2))])],
  nil,
  do(Now, add_friend(c3po, 'tcp://condor:9002'))).

commit([nil],
  [clock(Now), b([Now, not(alive(r2d2))])],
  nil,
  believe([Now, good('Princess Leia')])).

commit([nil],
  [clock(Now), b([Now, not(alive(r2d2))])],
  nil,
  believe([Now, alive(r2d2)]).

/* POLL */
commit([nil],
  [clock(Now), b([Now, not(alive(c3po))])],
  nil,
  kqml(Now, ['ask-all', sender(r2d2), receiver(c3po),
             content(b([Now, alive(c3po)])), language(prolog)]).

/* ASK */
commit([nil],
  [clock(Now), b([Now, alive(c3po)]), b([Now, not(wait_reply)])],
  nil,
  kqml(Now, ['ask-all', sender(r2d2), receiver(c3po),
             content(b([Now, good(_A)])), language(prolog)]).

/* WAIT */
commit([nil],
  [clock(Now), b([Now, alive(c3po)]), b([Now, not(wait_reply)])],
  nil,
  believe([Now, wait_reply] ).

/* REPLY */
commit(['ask-all', content(b([T, F]))],
  [clock(Now), b([T, F])],
  S,
  kqml(Now, [reply, receiver(S), sender(r2d2), content(b([T, F])),
             language(prolog)]).
```

Figure 5: *R2D2* -A Simple Agent-K Program.

```
/*BELIEVE */
commit( [reply, content(b([T, F]))], [], _, believe([T, F]) ).

/* OUTPUT TO LOG */
commit([reply,
content(b([T, F]))], [clock(Now)],
S,
do(Now, say(['I've been told ', [T, F], ' from ', S]))).
```

Figure 5: *R2D2* -A Simple Agent-K Program (continued).

The first two lines of the program define the agent's name, and provide a simple private action for output. The initialisation section consists of four commitment rules. The two private actions, *initialise_agent* and *add_friend* are required to establish communication links. They are defined by the interpreter. The address of an agent is defined by a Universal Resource Locator (URL) . We then establish the belief that Princess Leia is good, and finally the belief that *R2D2* is initialised (*alive*). All these rules will be triggered once, as the mental conditions only match if *alive* is false.

The next commitment rule, *poll*, repeatedly sends out a message to determine if *C3PO* is initialised. This is done until *C3PO* answers. The next two commitment rules, *ask* and *wait*, are triggered once *C3PO* is initialised. The first rule asks *C3PO* who is good. The second rule indicates that *R2D2* is expecting a reply from *C3PO*. This inhibits the *ask* rule being triggered again.

The final three rules are generic responses. *Reply* allows *R2D2* to reply to any question about its beliefs. It should be noted, however, that this does not respond exactly to the KQML *ask-all* message. Agent-K does not yet support a universal qualifier, therefore a Prolog predicate such as *setof* would have to be employed, together with a way of sending all the results sequentially. The *believe* rule simply allows *R2D2* to believe anything it is told. This is equivalent to the gullible behaviour of Agent-0. The final rule announces (to the user), any replies received by *R2D2*.

4.2 Commitment Rule Execution

Figure 6 shows an edited log of agent *C3PO* reacting to an incoming message from agent *R2D2*. It receives a KQML message via TCP/IP, and tests each rule against the message. It matches the *reply* rule, 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. Note that this is not shown in the log.

There are no rules that match. The agent then carries out its commitment to send the message via TCP/IP back to agent *R2D2*. In Figure 7 this is the last *reply* message received by *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 6: An Example Agent-K Log

4.3 Agent Interaction

The previous section showed one execution cycle for the agent *C3PO*. In this section we examine a trace of *R2D2*'s interaction with *C3PO*. This is shown in Figure 7 below. Incoming messages are shown as '<<<<', and outgoing messages as '>>>>'.

```

Time 1:
(ask-all :receiver c3po :sender r2d2 :content b([[10, 28, 94, 17, 2, 11], alive(c3po)]).) >>>

Time 2:
(reply :receiver r2d2 :sender c3po :content b([[10, 28, 94, 17, 2, 12], alive(c3po)]).) <<<<
(ask-all :receiver r2d2 :sender c3po :content b([[10, 28, 94, 17, 2, 12], alive(r2d2)]).) <<<<
BEEP BEEP! >[I've been told ', [[10, 28, 94, 17, 2, 12], alive(c3po)], ' from ', c3po].
(reply :receiver c3po :sender r2d2 :content b([[10, 28, 94, 17, 2, 12], alive(r2d2)]).) >>>
(ask-all :receiver c3po :sender r2d2 :content b([[10, 28, 94, 17, 2, 12], good(A)]).) >>>

```

Figure 7: An Example of Agent Interaction.

Time 3: (ask-all :receiver r2d2 :sender c3po :content b([[10, 28, 94, 17, 2, 13], good(A)]).)	<<<
(reply :receiver r2d2 :sender c3po :content b([[10, 28, 94, 17, 2, 14], good('Han Solo')]).)	<<<
BEEP BEEP! > [I've been told ', [[10, 28, 94, 17, 2, 14], good('Han Solo')], ' from ', c3po].	
(reply :receiver c3po :sender r2d2 :content b([[10, 28, 94, 17, , 2, 14], good('Princess Leia')]).)	>>>

Figure 7: An Example of Agent Interaction (continued).

Once initialised, *R2D2* sends a message to poll *C3PO*. *C3PO* responds and also sends a polling message. *R2D2* reacts by announcing its new belief that *C3PO* is alive. *R2D2* then sends out a message asking *C3PO* who is good. It also responds to the poll message from *C3PO* by replying that it (*R2D2*) is alive. In the final phase, *R2D2* receives two messages; the first is a question from *C3PO* asking who is good; the second message is the reply to *R2D2*'s question. *R2D2* announces this new knowledge to the user. Finally, *R2D2* responds to *C3PO* with the information that Princess Leia is good.

It must be noted that this is an extremely trivial example. However, several more complicated programs have been written using Agent-K. One was an implementation of a simple electronic commerce scenario, inspired by the Contract Net Protocol, Smith (1980). Another was a simple game of Battleships played between two agents. Agent-K is also being used as a testbed for experiments in theory revision between co-operating agents, Byrne (1995).

5 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.

5.1 AOP/Agent-0 Issues

The programming of Agent-0 is achieved through commitment rules which relate messages and mental conditions. These are similar to the 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. The use of traditional object-oriented methods for handling complex behaviours may provide more comprehensible code.

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. In addition, integrating Agent-0 and KQML has highlighted the difference between refraining and decommitting 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 decommitting; 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).

5.2 KQML Issues

We have noted several difficulties whilst using KQML. There are two categories of performatives for manipulating information. The first category is intended to be used with "knowledge-bases" (e.g. *insert*) and the second for transmitting "general" information (e.g. *tell*). However, the context in which one category should be used over another is not clearly defined. Another ambiguity appears with *unachieve*. Does this mean cancel an *achieve* request or undo an action already taken?

A second issue is that whilst there is a *:language* keyword for the *:content* field, an equivalent 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. A final plea would be for KQML messages to carry a time field. This would allow agents to make judgements about the recency of a request, and whether it had been superseded by the content of a message already received.

It should be noted that the KSE community currently appears to be divided over exactly how KQML should be specified; there are presently at least two competing syntaxes! In addition, a question has arisen over the ability of a single set of speech-acts to cover all domains. Indeed, we have encountered problems when agents have to interpret messages from other agents. It appears that the current set of KQML performatives should only be used for knowledge-base and database manipulation. For example, in applications such as calendar management, agents may want to communicate a request to schedule a meeting. This is arguably a specific speech-act, and may lead to confusion if, for example, *tell* is used (as different agents may interpret *tell* in different ways). We therefore suggest that each application domain should specify its own KQML performatives. *Tell* should be reserved for sending 'memos' to other agents, with no expectation that the other agent will even store the contents permanently.

Do such questions over the exact semantics of performatives invalidate the use of commitment rules? Should every message have a pre-defined response? Agent-K commitment rules allow an agent to have control over the actions performed. This is compatible with speech-act theory, i.e. a message does not guarantee action, it merely signifies what the sending agent would like to happen.

KQML does not currently specify the negotiation protocol an agent is using. We are not certain, but would expect the KQML layer to carry any negotiation information, such as whether an agent is using a Contract Net approach, Smith (1980), or a Distributed Problem Solving approach.

With regard to KAPI, we have not used the HTTP-based Agent Name Server, as we were unable to compile KAPI under the Solaris⁵ 2.0 operating system. This has also prevented us testing anything other than the TCP/IP transport mechanism. It should also be noted that there exists at least one other competing API. This is produced by UNISYS⁶. It operates in a different manner to KAPI in that it is interrupt driven, rather than providing a queue of messages.

5.3 Agent-K Issues

Creating an agent-based system using Agent-K appears more practical than with other implementations of Agent-0, as agents are able to communicate across a network, both with other Agent-K agents, and other KQML-based systems. However, such agents are far from perfect. Agent-K has many of the same weaknesses as the original Agent-0 language. For example, neither has a planning mechanism. In addition, the use of Prolog for beliefs and commitments restricts Agent-K agents to interaction with other Prolog-based agents. Whilst using KQML means that our agents can communicate with other agents, this does not mean that they can understand the *:content* field. This could be addressed by using KIF, Patil (1992), either as an intermediate language, or as the internal agent language.

There are some minor drawbacks associated with creating agents using 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.

5.4 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. As part of this effort we expect to begin work on defining an ontology of learning, or at least a new subset of KQML performatives that reflect communication between learning agents. For example, an agent may wish to test a proposition it has learned, or have a proposition revised by another agent, or it may wish to tell another agent that a rule has a certain classification accuracy. An agent may also instruct another agent to learn a set of classification rules on its behalf. Most learning algorithms rely upon concepts such as

⁵Solaris is a trademark of Sun Microsystems.

⁶No further details were available when this report was produced.

training examples, background knowledge, a learning result, and so on. These concepts might provide the basic set of primitives to be described by an ontology of learning.

Our second goal 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 language (such as planning and intentions). Finally, we plan to construct a simple user interface agent which can instruct Agent-K agents using KQML.

6 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 Rob Holton for comments on an earlier draft of this document, and to Dave Bayer and Alex McCall for struggling to evaluate Agent-K as part of their MSc.

7 Bibliography

- Brownston (1985)** L. Brownston et. al., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, 1985.
- Byrne (1995)** C. Byrne, *Refinement in Agent Groups*, In Proceedings of the Workshop on Adaptation & Learning in Multi-Agent Systems, IJCAI'95, Montreal, 1995.
- Davies (1995)** W. Davies & P. Edwards, *Agent-Based Knowledge-Discovery*, In Working Notes of the AAAI-95 Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments, AAAI Press, 1995.
- 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.
- Searle (1969)** J. R. Searle, *Speech Acts: An Essay in the Philosophy of Language*, Cambridge University Press, Cambridge, England, 1969.

Agent-K: An Integration of AOP and KQML

- Shoham (1990)** Y. Shoham, *Agent-Oriented Programming*, Technical Report No. STAN-CS-90-1335, Computer Science Department, Stanford University, 1990.
- Smith (1980)** R. G. Smith, *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver*. IEEE Transactions on Computers, C-29(12), pp 1104-1113, 1980.
- Thomas (1993)** S.R. Thomas, *PLACA, an Agent Oriented Programming Language*, Ph.D. Dissertation, Computer Science Department, Stanford University, 1993.