The University of Bologna Laboratory for Computer Science Research Technical Reports are available via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS in compressed PostScript format. Abstracts are available from the same host in the directory /pub/TR/UBLCS/ABSTRACTS in plain text format. All local authors can be reached via e-mail at the address last-name@cs.unibo.it.

**UBLCS Technical Report Series**

93-7  *Distributed Programming with Logic Tuple Spaces*, by P. Ciancarini, April 1993.
93-8  *Coordinating Rule-Based Software Processes with ESP*, by P. Ciancarini, April 1993.
93-14 *Guard Compilation in Logic Shared Dataspace Languages*, by M. Garpari, June 1993.
Symbol-level Requirements for Agent-level Programming

Mauro Gaspari\textsuperscript{1}  Enrico Motta\textsuperscript{2}

Technical Report UBLCS-94-2

February 1994

Abstract

In this paper we discuss the notion of programming at the knowledge level in the context of distributed AI systems. In particular, we show that although a number of languages defining abstract communication primitives have been proposed in the past few years, knowledge-level programming can only be supported if specific symbol-level requirements are satisfied. To illustrate this problem we formalize a subset of the KQML language and we show that the resulting synchronous architecture exhibits starvation and deadlock problems, which can only be solved if a careful definition of the underlying transport level is given. To this purpose we show how an alternative, asynchronous architecture can be defined for the same set of communication primitives, which avoids the aforementioned problems. Finally, we indicate a number of issues which require further investigation.

\textsuperscript{1} Mauro Gaspari, Dipartimento di Matematica, Università di Bologna, Piazza Porta S. Donato, 5, Bologna, Italy, email: gaspari@cs.unibo.it, Tel: +39 (547) 642813, Fax: +39 (547) 610100.
\textsuperscript{2} Enrico Motta, Human Cognition Research Laboratory, The Open University, Walton Hall, Milton Keynes, UK, email: E.motta@open.ac.uk, Tel: +44 (908) 653506, Fax: +44 (908) 653169
1 Introduction

In multi-agent architectures a set of autonomous processes (agents) cooperate to achieve a common goal. In general, these architectures can be seen as hybrid systems, in which each agent is characterised by its own representation mechanisms extended with communication primitives. Recently, two approaches have been proposed to support interaction between agents, which are both based on the speech act theory.

The first one is the Agent-Oriented Programming (AOP) framework proposed by Shoham [Sho93] to support multi-agent interactions. AOP specializes object oriented programming by defining the state of an agent, and restricting the kinds of messages which agents can send or receive. The state of an agent consists of beliefs, commitments, capabilities, and decisions. An agent can receive and send a number of specific messages, drawn from the speech act theory, such as ‘inform’, ‘request’, ‘offer’, ‘accept’, and others.

A different framework for agent-level programming is provided by KQML [Fin92], [Fin93], which consists of a set of communication primitives - called performatives - which aim to support “interoperability among intelligent agents in a distributed application” [Pat92]. The KQML performatives enable agents to exchange and request knowledge, and to cooperate during problem solving. The idea is that specific cooperation strategies, such as Contract Net [Smi88], can be emulated by writing programs which use the appropriate KQML performatives.

Both AOP and KQML follow an approach similar to the one adopted in distributed computing by a family of languages called coordination languages [Car92]. These extend sequential languages with constructs to support concurrency and coordination. In a similar way AOP and KQML extend knowledge representation formalisms with knowledge communication primitives, and focus on defining knowledge coordination languages, which can be used to specify a range of cooperation strategies. This approach is different from other ones, such as [Sin91], which focus instead on how beliefs and knowledge are modelled inside an agent. For instance, while KQML provides a tell primitive whose semantics is to inform other agents that a sentence is present in the virtual knowledge base of an agent, neither the knowledge representation language of the agent, nor the language in which the sentence is transmitted are specified a priori.

Although knowledge coordination languages can be seen as analogous to coordination languages in distributed computing, they are situated at a higher level of abstraction, as they support coordination not at the symbol but at the knowledge level [New82]. That is, they provide communication primitives which support the use, request, and supply of knowledge independently from implementation-related aspects. We can therefore see both AOP and KQML as languages for knowledge level, concurrent programing.

Both AOP and KQML describe the syntax and the structure of the communication primitives; neither, however, formulates precise requirements on the underlying concurrent system. Although AOP is based on a well known model of concurrent computations, the actor model [Agh92], the description in [Sho93] does not discuss the mapping between AOP and an underlying actor based language. Likewise, KQML designers only make some quite general assumptions concerning the ability of the underlying transport level to convey ordered and reliable point to point message passing. Moreover, in [Fin93a] they argue that reliability at the transport level “is less useful than it may appear, unless there is a guarantee of agent reliability as well. Such a guarantee is a policy issue and may vary among systems”. Finally, while the papers describing the AOP model discuss a

---

3 For example the TCP/IP protocol satisfies these assumptions.
number of requirements on the underlying languages - for instance that they should be able to support multiple levels of abstractions [Sho93], papers describing KQML do not outline any symbol level requirements which need to be obeyed to support these agent level approaches. For example whether the transport level is characterised by synchronous or asynchronous mechanisms or whether it makes use of blocking or non-blocking primitives.

In this paper we analyze whether or not symbol-level issues should be taken into account when designing knowledge coordination languages. That is, we try to answer the following two questions: i) Is it possible to program solely at the level of a knowledge coordination language? If this is the case, ii) what symbol-level requirements need to be taken into account?

We argue that although knowledge level distributed programming is possible, this is only the case if a number of careful assumptions on the communication primitives and the underlying symbol level are made. In particular, we show that the simple assumptions both AOP and KQML make about the transport layer are not sufficient to guarantee the feasibility of knowledge level programming. We also outline the architecture of an asynchronous system, and we show that it satisfies the given requirements for knowledge level programming for a subset of KQML.

The paper is organized as follows. In section 2 we discuss a number of requirements which characterise the notion of knowledge-level distributed programming. In section 3 we give the operational semantics of a synchronous model of a subset of KQML, and we show that it does not fulfil our requirements for knowledge-level programming. In section 4 we outline an alternative, asynchronous version of KQML, and we show that it avoids the pitfalls of the synchronous one. This asynchronous model is still too restrictive for practical purposes and therefore in section 5 we show how it can be enhanced without violating the given requirements. Finally, in sections 6 and 7 we discuss related approaches, the significance of our work for the area of multi-agent systems, and we outline a number of issues which are still open.

2 Requirements for Knowledge Coordination Languages

Our main aim in this paper is to analyse the communication primitives used in agent languages and the implications on the underlying levels (symbol-level) to see whether they can really support the notion of distributed, knowledge level programming - i.e. whether agents can be programmed solely at the speech act level. More precisely, we postulate that, in order to support this notion of distributed knowledge-level programming, an agent system should satisfy the following requirements.

i. The language should provide only knowledge level primitives.

ii. The programmer should not have to handle communication deadlocks\(^4\) explicitly. A deadlock situation occurs when two agents mutually wait for each other to answer a query [Sig89].

iii. The programmer should not have to handle starvation issues explicitly. A situation of starvation arises when an agent’s request for knowledge never gets executed despite being enabled.

iv. The programmer should not have to handle resource management issues.

---

\(^4\) Two types of deadlocks have been discussed in the literature: resource deadlocks, which can occur when two processes try to access a common resource, and communication deadlocks, which can occur in message passing - for example when the communication channels are bounded.
The first requirement is quite obvious and states that in order to talk about knowledge-level programming, the primitives need to be defined at the knowledge-level. This requirement appears to be satisfied by both KQML and AOP.

Requirements ii, iii, and iv are symbol level requirements, which enforce the idea that ‘true’ knowledge-level programming is concerned with the use, request, and supply of knowledge, and not with lower level issues such as the synchronization of competing requests, or the allocation of resources. In section 3 we will show that even a very simple subset of KQML doesn't necessarily satisfy them.

In this paper we make the assumption that our set of agents is statically defined and allocated on a set of processors, and that agents cooperate by means of a cooperative problem solving model - i.e. they carry out their own computational processes and exchange knowledge. This means that we do not consider here problems related to the satisfiability of requirement (iv), which concern the creation of new agents and the allocation of agents on a set of processors.

3 A synchronous KQML Model

The set of performatives used in KQML is large, as different kinds of coordination models are supported [Fin93a]. This means that it is difficult to provide a formal account of the whole system. Therefore we will only consider a subset of the KQML performatives (insert, ask, and tell) in a particular coordination context, i.e. one in which the recipient of a message is explicitly specified. We consider only communication actions and we do not deal with other aspects related to communication such as the contents of a message, the language used for representing messages, or the adopted ontology [Pat92]. Moreover, we do not model the agents' virtual knowledge base and we assume that each communication action has associated its contents in a given knowledge representation formalism. The model is based on synchronous communication and on a blocking ask primitive. We proceed in a formal way describing the operational semantics, giving a formal definition of the requirements, and showing that they are not satisfied in this system. Such an operational description is essential in order to have a rigorous definition of the communication mechanisms and as a basis to prove properties about them. Moreover, it clarifies the mechanisms on which a system is based and provides a precise specification for implementors.

3.1 Operational Semantics of a Synchronous KQML Model

We start by defining an operational semantics for this subset of KQML. Here we follow the structural transition system approach (SOS) [Plo81], which describes the behaviour of an agent by means of a transition relation on system configurations.

Agents are modelled as processes composed of a name, a set of programs, and a state. Agents react to messages received from other agents and from the user. Programs specify the way in which agents deal with messages. This model is similar to the concurrent object oriented programming model, where objects associate methods with messages [Mey93] and, more specifically, to the actor model described informally in [Agh90], and formally in [Agh92].

Let \( a, a' \) be names of agents, and \( A \) a variable which ranges over names of agents. The set of messages that can be received from an agent \( a \) in our subset of KQML are the following: \( \text{assert}(a), \text{tell}(a), \) and \( \text{ask}(a) \). \( \text{assert}(a) \) means that agent \( a \) is asked to insert a proposition into its virtual knowledge base (the KQML insert); \( \text{ask}(a) \) means that agent \( a \) is asked to prove a goal; and \( \text{tell}(a) \) means that an agent \( a' \) is sending an answer to \( a \). Each agent has an associated dispatch function which map the received message into the program which executes it.
As shown in the definition below a program can be either i) a communication action followed by a program, ii) a terminating internal program, or iii) a constant definition.

\[
\text{Program} ::= \text{CommAction.Progam} | \sqrt{\text{C}} \\
\text{CommAction} ::= \text{assert(A)} | \text{tell(A)} | \text{ask(A)}
\]

The symbol \(\sqrt{\text{C}}\) represents a terminating internal program - i.e. a program which does not involve communication actions and terminates. A constant \(\text{C}\) is an agent program whose meaning is given by a defining equation \(\text{C} = \text{def Program}\). This mechanism is useful as it allows us to define agents with infinite behaviour [Mil89].

As an example let's consider the agent program defined by the following two equations:
\(\text{C}=\text{ask(a).C'}\) and \(\text{C'}=\text{assert(a'').C}\). This program generates the following, infinite sequence of communication actions: \(\text{ask(a).assert(a'').ask(a)……}\)

The state of an agent can be:
- **busy**: the agent is executing a program;
- **idle**: the agent is idle;
- **wait(A)**: the agent is waiting for an answer from agent A.

Agents can receive and execute messages only if they are idle or waiting for an answer. There are no explicit receive primitive inside programs. Idle and waiting agents repeatedly look for messages and execute them according to the program definition determined by the dispatch function.

We define the set of configurations for our synchronous KQML model as follows. Let \(\text{A, A'}\) agent name variables, and \(\text{P}\) an agent program.

\[
\Gamma_{KQML} = P(\Gamma_{IDLE} \oplus \Gamma_{BUSY} \oplus \Gamma_{WAIT}) \\
\Gamma_{IDLE} = \{<A> | \text{A is an agent name}\} \\
\Gamma_{BUSY} = \{<A, P> | \text{A is an agent name and P a program}\} \\
\Gamma_{WAIT} = \{<A, P, \text{wait(A')}> | \text{A and A' are agent names and P is a program}\}
\]

The \(\oplus\) operator denotes set construction and it is also 'augmented' to denote set union. \(\Gamma_{IDLE}\) describes the set of agents which are idle, \(\Gamma_{BUSY}\) is the set of agents which execute programs, and \(\Gamma_{WAIT}\) is the set of agents which are waiting for an answer from another agent, \(\text{A'}\).

A transition system modelling KQML is a triple \((\Gamma_{KQML}, L, \rightarrow\) \[\{\rightarrow l | l \in L\}\)], where \(\Gamma_{KQML}\) is a set of configurations, \(L\) is a set of labels, and \(\rightarrow\) is the smallest transition relation satisfying a set of axioms and inference rules for each \(l \in L\).

In what follows we will introduce different sets of axioms and rules modelling different symbol level assumptions.

Let \(P, P',\) and \(P''\) be programs, \(L=\{\text{assert(A), tell(A,A'), ask(A,A'), }\sqrt{\text{C}}\}\) the set of labels, and \(\Gamma \in \Gamma_{KQML}\) a set of agents. The following set of labelled axioms and rules describe a system based on point to point synchronous communication with a blocking ask primitive.
The axioms labelled \( \text{assert}(a',a) \) and \( \text{ask}(a',a) \) state that agent \( a \) can send an assert or an ask message to agent \( a' \) only if this is idle. The new program, \( P' \), associated with agent \( a' \), is determined by the dispatch function. The axiom labelled \( \text{tell}(a',a) \) is enabled only if there is an agent \( a' \) waiting for an answer from \( a \). The axiom labelled \( \sqrt{\_} \) states that an agent can always perform a terminating internal program independently. The last rule is introduced to deal with constant definition and states that the constant substitution is hidden in the transition system - i.e. there is no label associated with the constant substitution.

The dispatch function must satisfy the following properties:

1. \( \text{dispatch}(\text{assert}(A')) \) returns a terminating internal program: \( \sqrt{\_} \).
2. \( \text{dispatch}(\text{ask}(A',A)) \) returns a possibly infinite program containing only one tell communication action, \( \text{tell}(A) \), which is the answer. \( A \) indicates the sender. All the actions which precede the answer must be different from \( \text{ask}(A) \) otherwise an immediate deadlock is generated since ask is a blocking operation\(^5\).
3. \( \text{dispatch}(\text{tell}(A'),P') \) takes as argument the program currently executed by agent \( A' \) and returns a possibly infinite new program which does not contain tell communication actions.

Let \( \gamma \in \Gamma_{\text{KQML}} \) and \( l \) a label ranging over the set \( L \). We say that a transition labelled \( l \) is enabled in \( \gamma \) if the corresponding axiom is enabled - i.e. there exists \( \gamma' \in \Gamma_{\text{KQML}} \) such that \( \gamma \twoheadrightarrow l \gamma' \).

\(^5\) This condition can also be relaxed as we will show in section 5.
We assume here that all the agents are initially idle, and the agent program is triggered by the user posing a query to the system (i.e. the user agent sending an ask message). A KQML computation is a possibly infinite sequence of labelled transitions:

\[ \gamma_0 \rightarrow^{l_1} \gamma_1 \rightarrow^{l_2} \gamma_2 \ldots \rightarrow^{l_n} \gamma_n \ldots \]

where \( \gamma_0 \subseteq \Gamma_{\text{IDLE}} \), \( \gamma_i \in \Gamma_{\text{KQML}} \), and \( l_0 = \text{ask}(A, \text{user}) \). We use the notation \( T(\gamma)_{\text{ask}(A, \text{user})} \) to denote the set of traces (i.e. sequences of transition labels) corresponding to the set of computations generated by the transition system with initial state \( \gamma \subseteq \Gamma_{\text{IDLE}} \). The \( i \)-th element of the trace \( t \in T(\gamma)_{\text{ask}(A, \text{user})} \) is denoted with \( t_i \). We say that \( l \in L \) is enabled in \( t \) if the transition labelled \( l \) is enabled in the corresponding computation.

The notation \( t \downarrow_a \) specifies the subsequence of the trace \( t \) containing the actions involving only agent \( a \), which can be of two different types:

- An infinite sequence of communication actions. This is the case when an agent program invoked by a dispatch operation generates an infinite computation. In this case the agent will never return in an idle state and therefore will not be able to answer queries.

- A finite sequence of communication actions followed by a terminating internal program. This is the case when all the agent programs invoke terminating computations. When an agent terminates its program it becomes idle and therefore ready to answer a new query.

### 3.2 Analysis of the Synchronous KQML Model

Let's now analyze the behaviour of this subset of KQML in order to see whether it satisfies the requirements given in section 2. The first problem to investigate is whether or not a situation of starvation can arise. This is trivially the case, as there is no scheduling mechanism for multiple ask or assert messages. As an example consider the following case where two agents, b and d, try simultaneously to execute an assert and an ask primitive with the same recipient:

\[
\begin{align*}
B &= \text{assert}(a).B \\
D &= \text{ask}(a).D \\
\text{dispatch}(\text{ask}(a,d)) &= \text{tell}(d).\sqrt{\phantom{V}} \\
\text{dispatch}(\text{assert}(a,b)) &= \sqrt{\phantom{V}}
\end{align*}
\]

Let's suppose we have a state \( \{<d,D>, <b,B>, <a>\} \) in which both the transition labelled \( \text{ask}(a,d) \) and the one labelled \( \text{assert}(a,b) \) are enabled.

\[
\begin{align*}
\{<d,D>, <b,B>, <a>\} &\rightarrow^{\text{ask}(a,d)} \{<d,D, \text{wait}(a)>, <b,B>, <a, \text{tell}(d).\sqrt{\phantom{V}}>\} \ldots \\
\{<d,D>, <b,B>, <a>\} &\rightarrow^{\text{assert}(a,b)} \{<d,D>, <b,B>, <a, \sqrt{\phantom{V}}>\} \ldots
\end{align*}
\]

Although they are both enabled, it is nevertheless possible that one of them is never executed. Starvation can be solved by enforcing a fair transition system (Francez, 1986), which guarantees that if a transition is sufficiently often enabled, it eventually gets executed. Several notions of fairness
have been defined, which give different interpretations of the enabling condition. For example "strong fairness" states that if a communication action is infinitely often enabled it will eventually be executed, while "weak fairness" states that if a communication action is almost always (always after a certain point) enabled it will eventually be executed. We believe that strong fairness [Fra86] is adequate for our agent systems\(^6\), while weak fairness does not prevent situations of starvation from occurring. This can be the case because some communication actions, such as ask, may be enabled an infinite number of times but only when the recipient agent is idle.

More formally a strong fair trace is defined as follows:

\[ t \in \Gamma \text{ask}(A, \text{user}) \]  

is a fair trace if \( \forall l \in L \) infinitely often enabled in \( t \exists n \in \mathbb{N} \) such that \( t_n=l \).

To ensure fairness individual agent computations must proceed and the implementation of the communication primitives must also guarantee reliability - i.e. whenever an assert message is executed it must eventually be received. This implies that the reliability requirement at the underlying levels is necessary to support knowledge level programming [Fin93a]. On the other hand most concurrent programming languages do not support fairness at the language level, as it is not trivial to implement a satisfactory fair mechanism - i.e. an implementation which guarantees at the same time fairness and non-determinism\(^7\).

In conclusion, if strong fairness is assumed, our agent system then satisfies requirement (iii). The other requirement that our system of KQML primitives needs to satisfy concerns communication deadlocks [Bar90]. Because the system is synchronous and the ask message is blocking, situations of deadlock can indeed arise. More formally a communication deadlock occurs in the system if \( \exists \gamma \tau a \) such that \( t \downarrow a \) is finite, \( \gamma \rightarrow \gamma' \), and \( a \) in not idle in \( \gamma' \). For example let's consider agents b and d; their programs are defined by constants B and D:

\[
B = \text{ask}(d), \text{assert}(a) . B  \\
D = \text{ask}(a) . \text{assert}(b) . D
\]

The computation deadlocks when the following state is reached:

\[
\langle b, \text{ask}(d), \text{assert}(a) . B \rangle, \langle d, \text{assert}(b) . D \rangle,
\]

agents b and d are not idle, and both the transitions labelled \text{ask}(d, b) and \text{assert}(b, d) are not enabled.

Another problem arises in this model because of the presence of agents potentially executing infinite computations; these agents may never return to an idle state. This implies that an attempt to send a message to such an agent can cause a deadlock.

Deadlock avoidance is therefore not guaranteed in this system. The problem cannot be solved with a careful implementation of the communication primitives because it depends on the blocking semantics of the primitives themselves and because agents may execute infinite loops. An alternative solution could avoid deadlocks by guaranteeing certain statically enforceable restrictions. For example an agent could be prevented from sending ask messages to agents with potentially infinite behaviour, and mutually recursive requests could be 'banned'.

---

\(^6\) This notion of fairness is also assumed in actor systems (Agha et al., 1992).

\(^7\) Most schedulers generate just one fair computation while, in general, a fair transition system contains a number of possible fair computations.
Different solutions to this problem have different advantages and disadvantages and a detailed discussion of this topic is outside the scope of the paper. The main point here is that our 'naive' model of a subset of KQML doesn't support the postulated requirements for knowledge-level programming. This suggests that the specification of knowledge-level communication primitives and agent-level languages has to go hand in hand with a precise specification of the underlying symbol-level architecture. To this purpose in the next section we discuss an alternative, asynchronous formalization of our subset of KQML, which satisfies our set of requirements and can therefore be seen as a candidate for the definition of a symbol-level support for agent-level programming.

4 An asynchronous system

4.1 Operational Semantics of an Asynchronous KQML Model

As discussed in the previous section our synchronous KQML model does not satisfy the given requirements for knowledge-level, distributed programming. This is due to the synchronous nature of the system and the blocking ask primitive. Let's now consider an alternative KQML formalization, which makes use of an asynchronous transport level based on a tuple space abstraction [Car92] and supporting a non-blocking ask primitive.

This execution model is slightly different from the previous one since an agent can now receive messages only when it is idle. Requests of knowledge do not block the agent executing them; it can continue its current execution. This implies that agents receive answers to ask questions only when they are idle.

A tuple space is used as repository of messages to support asynchronous associative communication and is usually modelled as a multiset\(^8\). Let TS a tuple space and + the union operator over tuple spaces (multisets). The use of such a communication medium does not cause the multi-agent architecture to have a central bottleneck since the tuple space can be partitioned among the different agents.

We define a new configuration for the asynchronous KQML augmented with TS. The set of agents \(\Gamma_{\text{WAIT}}\) is not included since the ask primitive is non-blocking.

\[
\Gamma_{\text{AKQML}} = P(\Gamma_{\text{IDLE}} \oplus \Gamma_{\text{BUSY}} \oplus \text{TS})
\]

\(\Gamma_{\text{IDLE}}\) and \(\Gamma_{\text{BUSY}}\) are defined as above. Let \(\Gamma \in \Gamma_{\text{AKQML}}\) a set of agents and \(L = \{\text{assert}(A), \text{tell}(A,A'), \text{ask}(A,A'), \text{get}(a), \sqrt{\}\}\) the set of labels. The asynchronous KQML transition relations are defined by the rules and axioms as follows:

\[\text{In the formal definition of the actor model described in (Agha et al., 1992) pending messages are also modelled as a multiset.}\]

8
The axioms labelled \texttt{assert(a', a)}, \texttt{ask(a', a)}, and \texttt{tell(a', a)} state that the agent \(a\) can always send an assert, an ask, or a tell message to agent \(a'\). The transition labelled \texttt{get(a)} is enabled if the agent \(a\) is idle and \(TS\) is not empty. The new program \(P'\) for agent \(a'\) is determined by the dispatch function. The axiom labelled \(\sqrt{\cdot}\) and the first rule have the same meaning as in the previous transition system. The last structural rule states that the transition modelling messages do not involve other agents in \(\Gamma\).

The properties 2 and 3 of the dispatch functions change for this asynchronous system:

1' = 1

2' \(\text{dispatch(ask(A',A))}\) returns a possibly infinite program containing only one tell communication action: \texttt{tell(A)}, which is the answer.

3' \(\text{dispatch(tell(A'))}\) returns a terminating internal program which does not contain communication actions.

In particular an agent \(a'\) which receives an ask message from an agent \(a\) can send an ask message to any agent including \(a\), before answering the query. This second ask message will not cause an immediate deadlock. In the case of a tell message, the dispatch function does not take a program as...
an argument and returns a terminating internal program, i.e. the result of a query can only change the internal state of an agent, not its program.

4.2 Analysis of the Asynchronous KQML Model

In contrast with the synchronous model described in section 3.1, the asynchronous KQML model fulfills the stated requirements.

Let’s consider starvation first. In order to avoid it, we need to ensure fairness in the transition system. Therefore the symbol level must be based on a reliable message delivery mechanism and a fair ‘fetch’ operation from the tuple space. This can be easily implemented by partitioning the tuple space into a set of queues managed by means of a FIFO discipline. This solution guarantees that requirement (iii) is satisfied in this system.9

Deadlocks cannot occur in the asynchronous system. This property can be trivially proven from analysing the rules in the transition system. These are always enabled provided that there is an action to be executed or a message to be retrieved. Of course property 2’ of the dispatch function must guarantee that a tell message is always executed as an answer to an ask message. Agents possibly involved in infinite computations do not constitute a problem since messages sent to them are simply ignored without causing a communication deadlock. Hence, requirement (ii) is satisfied in this system.

The proposed architecture satisfies the symbol level requirements for knowledge level programming but provides a weak version of the ask primitives. When an ask primitive is performed, the subsequent computation cannot benefit from the immediate answer. This could be a drawback for a distributed proof mechanism.

This problem can be solved by introducing a non-blocking ask primitive, and requiring that the agents be able to save the state of their computation. Thus, agents can handle recursive queries. Moreover, an agent can send an ask message, save its internal state, and start waiting for the next message received. This state-saving mechanism, together with the introduction of structures - such as queues - for storing and scheduling pending messages, ensures that our requirements are satisfied, without enforcing the restriction on distributed proofs which characterizes the asynchronous KQML model. We took this approach when specifying the communication mechanisms for the VITAL-KR [Gas93], an architecture supporting hybrid AI programming, which is part of the VITAL workbench for knowledge-base development [Dom93]. In [Gas93] we gave a formal description of a sequential version of the system, in the next section we provide a parallel account.

5 Communication in the VITAL KR

The parallel version of VITAL-KR we propose here makes use of the same KQML primitives as the two systems previously discussed and, like the asynchronous system described in section 4, is based on asynchronous communication which uses a tuple space abstraction as communication medium.

We define a VITAL-KR configuration as follows:

\[ \Gamma_{VKR} = P(\Gamma_{IDLE} \oplus \Gamma_{BUSY} \oplus TS) \]

\[ \Gamma_{BUSY} = \{<A,P,S> | A \text{ is an agent name and } P \text{ a program and } S \text{ a possible empty set of states whose elements have the following structure } (\text{id},P') \text{ where } \text{id} \text{ is an identifier and } P' \text{ represents the rest of the program} \} \]

---

9 Reliable message delivery cannot be guaranteed with certainty in an asynchronous communication model. We can assume an infinite buffer capacity and provide reliability with some level of confidence.
\[ \Gamma_{\text{IDLE}} = \{ \langle A, S \rangle \mid A \text{ is an agent name and } S \text{ is a set of states} \} \]

where \( \Gamma_{\text{IDLE}} \) is defined as above. Let \( + \) be both a multiset and a set union operator. Let \( \Gamma \in \Gamma_{\text{VKR}} \) a set of agents and \( L = \{ \text{assert}(A), \text{tell}(A, A'), \text{ask}(A, A'), \text{get}(a), \text{getid}(a), \sqrt{} \} \) the set of labels. We use the character \( '_' \) to denote an anonymous agent. The transition relations for the asynchronous VITAL-KR are defined by the following rules and axioms:

- \( \langle a, \text{assert}(a').P, S \rangle \oplus TS \rightarrow ^{\text{assert}(a', a)} \langle a, P, S \rangle \oplus TS + \{ m(a', \text{assert}(a')) \} \)

- \( \langle a, \text{ask}(a').P, S \rangle \oplus TS \rightarrow ^{\text{ask}(a', a', id)} \langle a, S + \{ (id, P) \} \rangle \oplus TS + \{ m(a', \text{ask}(a', a, id)) \} \)

  where \( id = f(a, S) \)

- \( \langle a, \text{tell}(a', id).P, S \rangle \oplus TS \rightarrow ^{\text{tell}(a', a, id)} \langle a, P, S \rangle \oplus TS + \{ m(a', \text{tell}(a', id)) \} \)

- \( \langle a, S \rangle \oplus TS + \{ m(a, msg) \} \rightarrow ^{\text{get}(a)} \langle a, P', S' \rangle \oplus TS \)

  where \( P' = \text{dispatch}(msg) \text{ and } msg \neq \text{tell}(A, _) \)

- \( \langle a, S + \{ (id, P) \} \rangle \oplus TS + \{ m(a, \text{tell}(a, id)) \} \rightarrow ^{\text{getid}(a)} \langle a, P', S \rangle \oplus TS \)

  where \( P' = \text{dispatch}(\text{tell}(a), P) \)

\[ \text{Figure 3: Vital-KR} \]

The axioms labelled \( \text{assert}(a', a), \text{ask}(a', a, id) \) and \( \text{tell}(a', a, id) \) state that an agent \( a \) can always send an assert, an ask, or a tell message to agent \( a' \). The identifier \( id \) is a function of an agent and its state uniquely identifying queries and answers. The transition labelled \( \text{get}(a) \) is enabled if the agent \( a \) is idle, if \( TS \) is not empty, and it is not a tell message, while \( \text{getid}(a) \) is enabled if the agent has received a tell message. The new program \( P' \) for agent \( a' \) is determined by the dispatch function, which, in the case of tell, takes as argument the program stored in the state identified by \( id \). This new set of axioms, together with the axiom for \( \sqrt{} \) and the two rules from the asynchronous model specified in the previous section, defines the transition system for the VITAL-KR.

The properties for the dispatch functions as defined as follows:

- \( 1" = 1 \)

- \( 2" \text{ dispatch}(\text{ask}(A, A', id)) \) returns a possibly infinite program containing only one tell communication action: \( \text{tell}(A, id) \), which is the answer.

- \( 3" = 3 \)

In particular the dispatch function must take into account the identifier while generating the answer.
The main advantage of this model over the one described in section 4 is that it not only satisfies our knowledge-level requirements, but it also provides a more powerful mechanism for supporting distributed proofs.

6 Related Work

The approach we have followed here to try and provide a more robust symbol-level grounding to knowledge-level, distributed programming has a lot in common with the actor model of computation [Agh90]. In the actor model the behaviour of an object is a function of incoming communication. Actors are self-contained agents that communicate by asynchronous message passing making use of three basic primitives: create, to create new actors; send-to, to send messages to other actors; and become, to change the behaviour of an actor. The actor model is situated at a lower level than the agent-level languages discussed in this paper. Nevertheless, there are some important similarities between the two execution models. Actors and agents have a unique mail address used to specify the recipient of a message. In both models there are no explicit receive primitives: as with the agents described in this paper, actors receive messages only when they are idle. The concept of transaction in an actor system, which is a request which generates subrequests, is also present in our system, where a query can generate a set of nested subqueries. The main difference between the actor model and ours is that the former does not allow non-terminating programs to be executed when answering a message. Moreover, the order of messages is not relevant in an actor-based system and, finally, actors do not provide symmetric primitives such as ask and tell. In summary the actor model provides a powerful set of primitives which can be used to build higher-level abstractions, in particular agent-level programming languages.

7 Conclusions

In this paper we have discussed a number of issues which we believe are important for multi-agent architectures. As far as we know, ours is the first attempt to provide a formal operational model of a speech act based language. We think this is a very important step in order to bridge the gap between distributed AI and semantic theories of concurrent systems. For example, future work could map the KQML set of performatives, or the VITAL-KR communication primitives, to the actor model defined by Agha, the CCS calculus [Mil89], or a coordination language, such as Linda [Car92]. There are three other aspects in this work which we think are relevant to multi-agent systems. Firstly, our proposal for a number of symbol-level requirements which need to be satisfied by knowledge-level, agent-based architectures. We see this as an initial step in line with the suggestion made by Allen Newell to investigate technical problems related to the knowledge level, in different AI contexts [New93]. Secondly, from a theoretical point of view it is a useful experiment on the adequacy of the chosen formalism (SOS) for describing this kind of complex systems. Thirdly, since KQML is still being defined, our analysis could provide useful feedback to its designers. Finally, a number of additional research items still need to be carried out. In particular, there is no guarantee that our list of requirements is complete. A more rigorous definition of the notion of knowledge-level programming could then enable us to be more rigorous in the definition of the related requirements.

Acknowledgements

Mauro Gaspari would like to thank Luca Aceto, David Murphy, Roberto Gorrieri, and Rune Gustavson for the interesting discussions about the topics presented in this paper.
Bibliography


Bibliography


