An Extensible Framework for the Development of Coordinated Applications

Enrico Denti, Antonio Natali, Andrea Omicini, Marco Venuti

LIA - DEIS - Università di Bologna
Viale Risorgimento, 2 – 40136, Bologna (Italy)
mailto:{edenti,anatali,aomicini,mvenuti}@deis.unibo.it
http://www-lia.deis.unibo.it/Staff/

Abstract. Distributed programming suffers from the lack of abstractions and tools required to handle and analyse the large amount of information characterising distributed systems. On the other hand, the separation of computation and coordination models definitely simplifies the design of a programming environment for distributed applications. Starting from this consideration, the $\mathcal{ACLT}$ coordination model extends the basic Linda kernel, by providing support for heterogeneous multi-agent systems, as well as for hybrid agent architectures integrating deduction and reaction. The design of the architectural support for the $\mathcal{ACLT}$ model led to the definition of a general-purpose scheme which is powerful enough to be used both for the system extension of the basic communication kernel and for building application-defined development tools. Such an approach is based on the idea of reactive communication abstractions, which can be programmed by agents according to a specification language which is rooted in the same model as the coordination language.

Keywords: Coordination models, Distributed programming environments, Extensible communication abstractions

1 Introduction

One of the main problems in developing distributed applications is the amount of information which has to be handled and analysed. Many aspects concerning concurrency, communication, synchronisation, sequential execution have to be taken into account altogether in order to get the whole picture of a distributed system. In particular, tools designed for the development of sequential applications do not suffice in a distributed environment: new tools, new abstractions are needed in order to cope with the complexity of distributed applications.

An effective approach to this problem consists in defining the *dimensions* of a distributed execution [9], where different *orthogonal* aspects are identified, each of which can be analysed independently and then combined so as to obtain the complete view of a system.

Coordination models like Linda [6] give then a relevant contribution to the definition of a framework for the development of distributed applications. With regard to this issue, two aspects are likely to have a strong impact:

i) the *separation* [8] between the computation and the coordination models;
ii) the concept of shared memory abstraction as a communication device, combined with the notion of *generative communication* [6].

According to (*i*), the aspects of communication can be considered separately from those of sequential execution. Thus, any computation model can in principle be integrated with coordination primitives, while retaining at the same time its basic operational semantics. Apart from the obvious conceptual economy of this approach, tools designed for the support of sequential programming are generally inadequate to capture distributed abstractions, but they can be re-used in order to describe the behaviour of the sequential components of a distributed system. As a further result, the definition of abstractions and tools for a distributed programming environment can now be concentrated on communication aspects only.

From the same viewpoint, the main consequence of (*ii*) is that communication channels can be re-interpreted as knowledge repositories, so that they can be used to get the whole picture of the communication state in a distributed application. With such an approach, one could build, for instance, a programming framework where a distributed application can be developed and executed keeping computation monitoring and communication monitoring separate.

$\mathcal{ACLT}$ (first presented in [12]) is a coordination model for multi-agent systems. Originating from research activities in the robotics field [15], $\mathcal{ACLT}$ is founded on a basic Linda-like kernel, extended with the notion of *logic tuple space* (see also [1, 4]) intended as the *theory of the communication*, along with the concept of *multiple tuple spaces* [7]. Tuples are first-order unitary clauses, and heterogeneous agents based on different technologies may exploit the logic theory represented by a tuple space according to their peculiar perception [5]. Non-logic agents (e.g. $\mathcal{ACLT}$-C agents) perceive the tuple space abstraction as a simple message repository, on which basic communication operations can be performed. Logic agents (e.g., $\mathcal{ACLT}$-Prolog agents), instead, can exploit logic tuple spaces as knowledge repositories which can be taken as bases for reasoning activities. For this purpose, the $\mathcal{ACLT}$ coordination language provides a small set of *demo* primitives allowing the design of hybrid agent architectures for logic agents, integrating deduction and reaction in a unique conceptual framework [12].

When facing the problem of providing a full-fledged programming environment for the $\mathcal{ACLT}$ model, two options are basically available:

- hardwiring the language and the supporting tools in an *ad-hoc* environment;
- defining a general-purpose extension scheme, such that the basic communication model can be enriched with new primitives (like the *demo* primitives), and the programming framework can be extended according to application needs.

The main aim of this work is to show how such a general-purpose scheme can be defined and exploited in a distributed programming environment like the $\mathcal{ACLT}$ supporting framework.

The proposed approach actually focuses on enhancing the communication abstraction. The basic idea is to shift the reactivity of the communication abstraction from the *communication state* to the *communication events*, lifting system observability from tuples to operations on tuples. Moreover, an event-(re)action scheme is defined allowing reactive activities to be associated to the basic communication operations. As a result, according to [11], *active* processes determine the behaviour of the distributed system, while *reactive* ones are in charge of the system monitoring.

Moreover, the supporting model provides a *specification language* meant to program the behaviour of the communication abstractions in terms of event reactions. Both supporting tools and language extensions can be built using this simple specification language, which is in turn based on the same communication model of the basic system: tuples and basic operations over tuple spaces. In particular, this work intentionally ignores the intricacies related to the definition of language extensions, by focusing instead on how such a specification language can be exploited to build supporting tools for distributed programming.

The paper is structured as follows. Section 2 discusses the role of the tuple space communication abstraction in a distributed programming environment like $\mathcal{ACLT}$. Section 3 introduces the general-purpose extension scheme upon which the $\mathcal{ACLT}$ multi-agent programming framework is built. This is then exploited in Section 4 to show how some simple support tools for the $\mathcal{ACLT}$ programming environment can be practically defined. Section 5 is devoted to final remarks and conclusions.

2 Enhancing the communication abstraction

From the viewpoint of the design of a distributed programming environment, the main consequence of the separation between the computation and the coordination models, discussed in [8], is that we can concentrate on the definition of abstractions and tools for communication only.

In the $\mathcal{ACLT}$ model, communication devices are represented by a multiplicity of named logic tuple spaces. A tuple space is a collection of first-order unitary clauses, uniquely identified by a ground term. So, the set of the logic tuple spaces can be read as the theory of the communication and taken as the view of the communication state in a distributed application.

However, such views are not enough for our purposes. In fact, they can provide static inter-process communication views [9] (i.e. views based on the content of the communication channel), showing the effects of interaction but not the interaction itself. Instead, it is widely acknowledged that distributed system monitoring calls for *dynamic* inter-process communication views, showing the *operations* performed on the communication channel. As a result, from the system viewpoint, the observability level should be moved from communication state observability to communication event observability. Since a logic tuple space can be seen as a communication channel view, the above requirement results in lifting the observability from tuples to operation over tuples.

Even though a tuple space may be perceived as a merely passive component, it *is* actually (implicitly) reactive, since it must be able, at least, to wake up suspended agents when a suitable new tuple is inserted. From this perspective, it seems natural to ground our general-purpose extension scheme on the *enhancement of the communication abstraction*, and in particular on the capability of specifying new tuple space behaviours.

Then, although a tuple space is intrinsically reactive to its state changes, as shown above, such a reactivity level is not enough, since it captures the effects of operations on tuple spaces, but not the operations themselves. So, primitives which do not result in a state modification, such as *read*, cause, by definition, no reaction at all.

Our approach is then to enhance the tuple space reaction level, making it sensitive to communication events rather than just to communication state changes. Although such an enhancement obviously concerns primarily the system level, the resulting ability should be made available also to the upper level of the application agents, thus providing the programmer with a flexible way to control the communication channel behaviour.

For this purpose, programmers should be provided with a *specification language* letting them specify tuple space reactions. For the sake of uniformity and conceptual economy, such a language should be based on the same model adopted for usual inter-agent communication, i.e. it should be expressed in terms of logic tuples and tuple spaces. The resulting architecture can be interpreted at two different abstraction levels: an *agent level*, and a *system level*, according to a model where all interactions, at both levels, can be modelled with the same basic coordination language.

3 An architecture for a multi-agent programming environment

As discussed above, in order to provide a general-purpose extension scheme for system programming, a notion of reaction to communication events has to replace the conventional, implicit notion of reaction to communication state changes. Moreover, a specification language is needed to capture the relevant events and to define the intended reactions of the communication devices.

3.1 The reaction model

The first idea of the proposed specification language is to provide the ability to define a set of logical events, each denoted by a unique name, and associated to physical events. Multiple physical events may correspond to the same logical event, and, conversely, multiple logical events may be associated to the same physical event.

Then, for each logical event, a set of distinct activities may be specified, which can be interpreted as independent, *reactive* agents. The activities of these agents, which are conceptually at a different level[1] from the application agents, are expressed in terms of a first-order logic coordination language, where only a subset of the coordination language available to logic agents can be used. For instance, a possible reaction might consist of a goal like `in_noblock(p(a)), in_noblock(p(X)), out(pp(a,X))`.

If multiple reactions are specified in response to a given logical event, they are all executed independently one from each other, in a non-deterministic way. In addition, any specified reaction is performed as an atomic action, thus providing a mechanism for expressing transactions. For instance, the example above produces either the deletion of two `p/1` tuples and the addition of one `pp/2` tuple, or it yields no effect at all.

Relevant events At the system level, the basic events to be intercepted are all those related to the communication primitives *in*, *out*, *read* and their variants.[2] However, *in* and *read* operations are conceptually different from *out* operations. In fact, while the latter simply adds a new tuple to a given tuple space, *in* and *read* can not conceptually be reduced to a single phase: rather, three distinct phases can be identified.

First, a tuple is sent by the agent to the tuple space. Then, the tuple space looks for a unifying tuple, and the agent possibly suspends its execution if no such tuple exists. Finally, when a unifying tuple is eventually found, the client agent is given such a tuple and subsequently resumes its execution.

While the second phase concerns only the internal tuple space behaviour, the first and the third phases represent two conceptually distinct events, which need to be intercepted separately. We will refer to the first phase, where only the tuple is considered, as the *pre* phase, and to the third phase, where only the unifying tuple is taken into consideration, as the *post* phase. Note, however, that since *out* operations imply no response from the tuple space, they can conceptually be thought of as performing the *pre* phase only.

Each communication event also features some natural properties, such as the name of the performing agent, the provided tuple, and the unifying tuple. Moreover, in a multiple tuple space environment such as $\mathcal{ACLT}$, the specification of the operation yielding a given reaction should involve the tuple space name, too.

[1] However, reactive agents do not constitute a meta-level, but rather a lower level with respect to the agent level. Actually, reactive agents can be conceived as a sort of kernel extension, allowing programmers to monitor typical kernel-level events, such as the occurrence of communication operations.

[2] The typical variants for *in* and *read* integrated in a logic language as Prolog are the ***non-blocking in*** and *read*, called also the ***predicate versions*** of the primitives [2]. These primitives (here, `in_noblock` and `rd_noblock`) replace the suspension semantics with the success/failure semantics, in that they fail (instead of causing the suspension of the performing agent) when no matching tuples exist.

The specification language Two kinds of special tuples (`map/2` and `react/2` tuples) constitute the proposed specification language, allowing the definition of

- the association between communication operations and logical events;
- the triggering of reactions in response to logical events.

This two-step specification allows multiple physical events to be associated to the same logical events and viceversa, multiple reactions can also be associated to the same logical event.

The first association is represented by a special tuple of the form

`map(`***`Operation, Event`***`)`

which captures the idea that each time the physical ***Operation*** is performed on the tuple space, a logical ***Event*** occurs.

If multiple ***Operations*** are bound to the same ***Event***, all the corresponding physical events will result in the same tuple space reaction. This could be useful, for instance, to build a simple tracer, in which case all operations should just be captured into the logical event `trace`, and handled somehow recording the performed operation.

If, on the other hand, one ***Operation*** is bound to multiple ***Events***, its occurrence will result in the triggering of a collection of logical events, which will be handled autonomously and asynchronously one from each other.

Reactive agents, or simply *reactions*, are specified through tuples of the form

`react(`***`Event, Goal`***`)`

where ***Event*** is the name of the logical event triggering the reactive agent, and ***Goal*** is the *body* of the reaction, that is the collection of primitive operations to be executed in order to perform the reaction.

Only a subset of the coordination primitives are allowed inside the body (***Goal***) of the reaction specification: in particular, suspensive operations (such as *in* and *read*) are obviously prohibited, due to the atomic action semantics of reactions. Instead, `out`, `rd_noblock` and `in_noblock` can be freely combined in a reaction body.

Moreover, it is not possible to nest goal demonstrations in ***Goal***, by forcing the proof of an atomic formula with respect to a given logic theory. Instead, specific primitives can be used in a reaction body in order to test the intrinsic properties naturally featured by each relevant event, such as the tuple space involved, the name of the agent which caused the event, the provided tuple, the unifying tuple, and the phase of the current operation (*pre* or *post*, when applicable). As a result, also the following primitives can be used inside a reaction body:

- `current_ts(?TS)`
- `current_tuple(?T)`
- `current_op(?Op)`
- `current_agent(?Ag)`

- **pre**
- **post**
- **success**
- **failure**

Obviously, **pre** and **post** primitives succeed only in the *pre* and *post* phases, respectively. Correspondingly, **current_tuple** returns the provided tuple in the *pre* phase, and the unifying tuple in the *post* phase. Moreover, **success** fails only in the *post* phase of a failed non-blocking primitive, in which case **failure** succeeds.

If multiple reaction goals are associated to one given logical event, a new *reactive agent* is conceptually activated for each reaction goal. Since such reactions are conceptually independent, such agents might work sequentially, concurrently or in parallel to the one other, depending on the underlying system.

Example 1 (A simple tracer). In order to show the power of this specification language, we show here how a very simple tracer could be specified, which intercepts all possible actions over the tuple space (either modifying or not modifying its state) turning them into a common *trace* logical event, which is then handled by generating a visible tuple showing the occurred physical event.

For the sake of simplicity, we avoid here using multiple tuple spaces: instead, all communication primitives[3] are refered to the default $\mathcal{ACLT}$ tuple space.

```
map(rd, trace).
map(rd_noblock, trace).
map(nd_rd, trace).
map(in, trace).
map(in_noblock, trace).
map(nd_in, trace).
map(out, trace).

react(trace, (pre, current_op(Op), out(happened(Op)))).
```

Reactions as transactions Reactions are conceived as *atomic actions*. If all the primitives constituting a reaction specification succeed, then the reaction is brought to an end, and the side-effect operations possibly associated to it are triggered all at once. Instead, if even only one subgoal fails, the reaction is cancelled, having no effect at all. Thus, at the agent level, reactive agents are perceived as featuring a transaction semantics.

Instead, from a system level viewpoint, reactions can be seen as sequences of operations. In particular, the primitives constituting the body of a reaction are conceptually executed as a sequence. So, the relative order of the subgoal in a reaction body may influence the result of a reaction. However, this has no influence over the agent level of perception since reactions are atomic actions at that level. In particular, in case many side-effect operations occur in a successful

[3] Primitives nd_rd and nd_in allow non-deterministic suspension on more than one tuple, following the example of the SICStus Prolog Linda library [13]

reaction, then active agents would perceive such events as happening all at the same time, and producing a single transition of the tuple space state.

A further consequence of the atomicity of reactions is that they cannot be nested. Since the effects of a reaction R are actualised if and only if R is successfully ended, no further reaction to the relevant events which could occur in R can be fired until R is over, and the corresponding state transition is completed.

Example 2 (A simple reaction). For instance, the code below defines a reaction specifying that any `out` performed over the tuple space `world` should transform the tuple space `pqTS` by adding a new `a/1` tuple and transforming a tuple of the form `p(`*t*`)` possibly contained in `pqTS` into a tuple `q(`*t*`)`, where *t* is a generic term.

```
map(out, outEvent).
react(outEvent, (current_ts(world), current_agent(A),
                 out(a(A))@pqTS,
                 in_noblock(p(X))@pqTS, out(q(X))@pqTS)).
```

In case a tuple of the form `p(`*t*`)` exists, the reaction is successfully finished, and the transformation is completed as a transaction: the tuple space `pqTS` changes its state (two tuples inserted, one tuple removed) in a one-step transition. If no suitable `p/1` tuple exists, the reaction cannot be completed, consequently it produces no effect at all over `pqTS`, and no state change takes place in the system. In particular, even the `out(a(A))` operation, conceptually performed before the `in_noblock(p(X))`, gives no results in the case of `in_noblock` failure.

Moreover, in case this reactive agent succeeds, it results in three relevant events: two `out` and one `in_noblock`. In case some reactions are specified for these events too, they would be executed only after this reactive agent has successfully completed its task, with no nested reactions.

3.2 An abstract architectural reference model

In this Section we present an abstract architectural reference model, which captures all the concepts, components and functionalities needed to implement the proposed architecture for the support of the $\mathcal{ACLT}$ model. However, this model should not be intended as an immediate and direct transposition of the practically implemented architecture, but, rather, just as a (possibly inefficient and unoptimised) way to model the system behaviours.

In the previous Sections we showed that a multi-agent system organised around a tuple space abstraction for communication, synchronisation and coordination could be observed at (at least) two different perception levels, the (higher) agent level and the (lower) system level, where the reactive aspects of the tuple space behaviour are implemented.

Therefore, our architectural model should include a general-purpose scheme to support a flexible tuple space abstraction, where the same coordination model may conceptually be exploited both at the agent and at the system level. This means that the coordination abstraction should be designed, at some level, as an open kernel, with mechanisms allowing default behaviours and policies to be programmed so as to trigger new asynchronous (re)actions.

Unlike mechanisms typically used in the sequential programming language area, (such as traps, for low-level languages, and late binding, in higher-level languages like C++), which aim to support the design of open application agents, we require here that such a degree of openness is provided by the tuple space abstraction, thus transferring to the coordination device the responsibility of ensuring the necessary system flexibility.

Since the tuple space reactions are represented in the specification language described in the Subsection 3.1, which is based on the two special tuples **map** and **react**, the first issue is *where* such specifications should be stored. For this purpose, one of the tuple spaces is chosen as the *Specification Tuple Space* (*SpecTS* in the following), where specification tuples are deposited by agents by means of a series of **out** operations.

Since specification tuples are special only because of the way they are interpreted, we should determine which component of the system is in charge of such an interpretation. To this end, the concept of a special *system agent* is required. The system agent is notified of all events and - when needed - triggers the specified reactions. Then, each time a basic tuple space event occurs, the agent should check the specifications, translate it into the logical events bound to it, and activate as many reactive agents as the specified **reacts**. Notice that, because of the restricted language allowed in reaction goals, no synchronisation is possible between reactive agents, or between reactive agents and active (application) agents.

However, in order to trigger the specified behaviours, all tuple space events must somehow be made observable. With regard to this issue, everything goes as if the occurrence of any relevant event were matched by the insertion of an explicit *service tuple* in a special tuple space, called *Service Tuple Space* (*ServTS* in the following), visible only at the system level. A service tuple represents the operation to be performed along with all its properties, such as the operation phase (*pre* or *post*), the performing agent, the tuple space involved, and so on. This conceptually captures the new notion of observability of the communication processes, which is shifted from the communication state to the communication events, by using the same metaphors as the basic model.

Moreover, observability of the communication events is not achieved by intercepting and then reflecting such events to a meta-level which is in charge of their handling. Rather, it is obtained by coupling any occurrence of a relevant event with its representation in the form of a special tuple, which is directly perceived only at the level of the reactive agents (which is conceptually a lower level than the agent level). In this way, while the execution of a communication operation may trigger a multiplicity of further (asynchronous) computational threads in the form of atomic reactions, the behaviour of the communication operation itself remains unchanged. As a result, this model does not introduce any direct interference with the basic Linda coordination kernel.

For instance, the request for `in(p(X,Y))@ts` operation performed by an agent ag could be modelled by a service tuple of the form `service(in, pre, ag, ts, p(X,Y), ...)` put in the *ServTS*, representing to the *pre* phase of the `in` operation. Then, suppose the application agent `ag` suspends itself waiting for the answer: when it is finally awakened, it receives the resulting tuple (for instance, `p(a,6)`). This will be represented by the eventual appearance of a service tuple like `service(in, post, ag, ts, success(p(a,6)), ...)` in the *ServTS*, corresponding to the *post* phase of the `in`.

A major point here is that the *ServTS* should be thought as being conceptually different from the agent tuple spaces, since it is a conventional, state-reactive tuple space, with no event-reaction capabilities at all. If it were not so, the reaction handling mechanism would enter an endless loop, as the output of the service tuple would trigger a new event, which should be handled using the same protocol.

The system agent may then be considered as continuously waiting for service tuples in the *ServTS*, searching the *SpecTS* for specifications about the performed operation, and possibly forcing the execution of some reactive agents. This conceptual behaviour shows that reactive agents do not merit the full application agent status, since they are nothing more than mere asynchronous execution threads, like Shared Prolog [1] ephemeral agents. This is reflected, for instance, also by the fact that they don't even have a name: actually, a `current_agent` subgoal within the reaction goal would return the name of the application agent performing the tuple space operation.

Finally, it is worth noting that the conceptual architecture sketched here does in no way imply a centralised approach to the implementation of the $\mathcal{ACLT}$ distributed programming environment.

4 Building support tools

The general-purpose event-reaction mechanism proved to be effective in the design of programming support tools for the $\mathcal{ACLT}$ environment, which have been exploited also during the development of the $\mathcal{ACLT}$ system itself.

In order to monitor a distributed application effectively, one needs a method to watch the contents of a given tuple space (say, `world`) and to trace the operations performed by the application agents. These requirements have been met in the $\mathcal{ACLT}$ environment by providing two support tools: the *visualiser*, and the *tracer*.

4.1 The visualiser

Given that tuple spaces represent *views* of the inter-process communication state, the first issue is to provide a simple tool for the *visualisation* of the contents of a tuple space [9].

The $\mathcal{ACLT}$ visualiser is a tool designed to provide a dynamic mirroring of the contents of a given tuple space, and is built as an agent which monitors the contents of a tuple space and updates a visualisation (such as a terminal window) whenever the tuple space is modified. As could be expected, such behaviour can be achieved by suitably programming the tuple space reaction, so that a proper signal is raised whenever a tuple space modification event occurs. As a result, the visualiser can be designed as an endless loop waiting for such a signal, which is handled by correspondingly updating the visualisation.

Then, only two reactive behaviours (agents) are needed, one for each basic category of operations which can modify the tuple space content:

- a logical event to intercept *in* operations;
- another logical event to capture the single *out* operation.

Both events result in the emission of a tuple of the form

`do(`***`Command`***`(`***`Tuple`***`))`

telling the visualiser how to update the visualisation, into a separate tuple space **visTS**, used as a knowledge repository by the visualiser agent.

A possible event reaction specification needed for this purpose is the following:

```
map(out, outEvent).
map(in, inEvent).
map(nd_in, inEvent).
map(in_noblock, inEvent).

react(outEvent, (
   current_ts(world), current_tuple(T), out(do(add(T)))@visTS)).
react(inEvent, (post,
   current_ts(world), current_tuple(T), out(do(del(T)))@visTS)).
```

Although the specified reaction is similar in both cases, the one related to **inEvent** is constrained to be performed only in the *post* phase, when the matching tuple to be removed has been identified. Moreover, since the visualisation is not influenced by knowledge access operations (such as *read*), these primitives are not considered as relevant events.

Operationally, when activated over a given tuple space, the visualiser installs its event-reaction specification tuples in the *SpecTS* by means of a series of out operations. Then, it reads[4] and visualises the whole current tuple space content, and enter the endless loop waiting for do/1 tuples.

As a further result, once a distributed tuple space implementation is provided, the same mechanism used to design the visualiser can be taken as a basis to build an error recovery mechanism. By ensuring that the monitored and the visualiser tuple spaces are physically allocated on different machines, the "private copy" of the tuple space content maintained by the visualiser could be exploited to recover from a crash of the monitored tuple space, thus effectively enhancing the system robustness.

[4] Through the $\mathcal{ACLT}$ rd_all primitive.

Example 3 (Visualiser). To show the previously described visualiser working, let us consider a very simple multi-agent system, where three agents **emy**, **ely**, and **evy** communicate through a (initially empty) tuple space **world**. Suppose the system behaves as follows:

- agent **emy** suspends itself on tuple space **world** with a blocking **in(a(X))**;
- subsequently, agent **ely** performs two operations on the tuple space **world**: first, an **out(b(1))**, then, a non-blocking **in_noblock(a(1))**.
- finally, agent **evy** performs two more operations on the tuple space **world**: first, an **out(a(2))**, then, a non-blocking **rd_noblock(b(Y))**.

The visualiser output is then determined by the following command tuples, recorded in tuple space **visTS**:

```
do(add(b(1))).
do(add(a(2))).
do(del(a(2))).
```

A careful implementation of this model, retaining the correspondence between the relative ordering of both agent level and system level events, would make tuple ordering in the **visTS** tuple space relevant, by implicitly providing some information about the temporal succession of operations.

4.2 The tracer

The simple visualisation of the tuple space content may be insufficient when analysing the correct behaviour of a set of agents and of their interactions. In fact, this would provide no information about knowledge access operations (i.e., *read*), nor would it allow non-blocking *in* operations, which may fail without removing any tuple, to be always detected.

Therefore, a different inspection tool is required, which monitors all tuple-space-related events. Again, such a tool should specify a set of proper event reactions, aimed to turn all operations into visible tracing tuples, such as the following:

```
map(out,        preEvent).
map(rd,         preEvent).
map(in,         preEvent).
map(nd_rd,      preEvent).
map(nd_in,      preEvent).
map(rd_noblock, preEvent).
map(in_noblock, preEvent).
map(rd,         awakeEvent).
map(in,         awakeEvent).
map(nd_rd,      awakeEvent).
map(nd_in,      awakeEvent).
map(rd_noblock, postEvent).
map(in_noblock, postEvent).
```

```
react(preEvent, (pre,
  current_op(Op), current_ts(Ts), current_tuple(T), current_agent(Ag),
  \+ T = trace(_,_,_,_),
  out(trace(agent(Ag), performs(Op),
      on_tuple(T), on_ts(Ts)))@trTS)).
react(awakeEvent, (post,
  current_op(Op), current_ts(Ts), current_tuple(T), current_agent(Ag),
  out(trace(agent(Ag), sleeping_on(Op),
      awaken_with(T), on_ts(Ts)))@trTS)).
react(postEvent, (post, success,
  current_op(Op), current_ts(Ts), current_tuple(T), current_agent(Ag),
  out(trace(agent(Ag), asking_for(Op),
      succeeds_with(T), on_ts(Ts)))@trTS)).
react(postEvent, (post, failure,
  current_op(Op), current_ts(Ts), current_tuple(T), current_agent(Ag),
  out(trace(agent(Ag), asking_for(Op),
      failed, on_ts(Ts)))@trTS)).
```

Here multiple logical events map the same physical event, in the same way as multiple physical events are mapped into the same logical event. For instance, an `in` operation is mapped both into a logical `preEvent` (which is actually executed only in the *pre* phase) and a logical `awakeEvent` (to be actually executed only in the *post* phase). In turn, the `preEvent` logical event maps practically all physical operations.

In order to provide a diagnostic tracing which is as expressive as possible, such mappings are closely tailored to the specific operation: so, for instance, two *post* events are provided, one for possibly suspensive operations and another one for non-suspensive operations with a success-or-failure semantics. An example could be an `in_noblock` operation, which is never suspensive, and triggers a more appropriate `postEvent` in its *post* phase rather than an `awakeEvent`. Notice the check in `preEvent` which is aimed to avoid tracing the trace tuples themselves, which would cause an endless loop.

Obviously, since tracing information is collected in the `trTS` tuple space, a visualiser like the one described in Subsection 4.1 might be used to make the tracing tuples visible.

Example 4 (Tracer). To show the previously-described tracer working, let us consider again the very simple multi-agent system introduced in the Example 3 on page 12. The tracer output results in the following sequence of tracing tuples in the tuple space `trTS`:

```
trace(agent(emy), performs(in),
                  on_tuple(a(X)), on_ts(world)).
trace(agent(ely), performs(out),
                  on_tuple(b(1)), on_ts(world)).
trace(agent(ely), performs(in_noblock),
                  on_tuple(a(1)), on_ts(world)).
trace(agent(evy), performs(out),
                  on_tuple(a(2)), on_ts(world)).
```

```
trace(agent(emy), sleeping_on(in),
                  awaken_with(a(2)), on_ts(world)).
trace(agent(evy), performs(rd_noblock),
                  on_tuple(b(Y)), on_ts(world)).
trace(agent(evy), asking_for(rd_noblock),
                  succeeds_with(b(1)), on_ts(world)).
```

As already noted in the Example 3, the tuple ordering in the `trTS` tuple space may be relevant, according to the peculiar implementation of the model, thus implicitly provide information about the temporal succession of traced operations.

4.3 Further remarks

The event-(re)action scheme defined in the previous Sections has also been exploited to extend the basic $\mathcal{ACLT}$ language. In particular, it has been used to provide a family of hybrid *demo* primitives [12] allowing reasoning activities over the evolving logic theory represented by a logic tuple space. The discussion of how such a complex system extension is achieved is outside the scope of this paper, and therefore will not be reported here.

5 Conclusions

In this work we have presented a framework for multi-agent system programming, based on the logic tuple space abstraction, which provides a notion of event reaction to tuple space operations which can be seen as the specification of low-level, asynchronous reactive agents.

One of the closest models known in the literature is represented by Shared Prolog, or ESP (Extended Shared Prolog) [1, 4, 3], whose main differences from $\mathcal{ACLT}$ are the following.

First, the neatness of the ESP model is by far superior to that of $\mathcal{ACLT}$, since ESP keeps sequential execution threads and communication operations clearly separate [3]. Instead, $\mathcal{ACLT}$ makes no syntactic restriction over the sequence of the operations performed by both active and reactive agents: even though this results in a less clean scheme, it seems more adequate to application environments (such as robotics [15]) where hybridness is a requirement [10] at any abstraction level.

Moreover, ESP agents are ephemeral, and express only reactive activities, while $\mathcal{ACLT}$ provides a framework where both active and reactive agents can be designed and combined.

Finally, ESP active tuples can be used only to model static inter-process communication views, since they react only to tuple space state modifications. Instead, the $\mathcal{ACLT}$ programming framework provides a specification language whose special tuples can also be used to model dynamic inter-process communication views, by allowing reactions to operations over tuple spaces to be captured, too.

This reaction mechanism has proved flexible enough to easily support the construction of effective support tools for system programming, which are needed to develop real distributed applications. Moreover, it has been used to extend the $\mathcal{ACLT}$ coordination language with hybrid primitives integrating deduction and reaction [12].

The $\mathcal{ACLT}$ multi-agent programming environment presented here has been mainly implemented on top of the SICStus Prolog system [14], and is currently working in a network of Sun, HP and Linux workstations. The actual implementation proved to be quite robust, and will probably be tested in advanced real application environments, in order to verify the effectiveness of the $\mathcal{ACLT}$ model.

References

1. A. Brogi and P. Ciancarini. The concurrent language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1), January 1991.
2. N. Carriero and D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.
3. P. Ciancarini. Coordinating rule-based software processes with ESP. Technical Report UBLCS-93-8, Laboratory of Computer Science, University of Bologna, April 1993.
4. P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12, 1994.
5. E. Denti, A. Natali, A. Omicini, and M. Venuti. Logic tuple spaces for the coordination of heterogeneous agents. In *Proceedings of the First International Workshop "Frontiers of Combining Systems", FroCoS'96*, Munich, Germany, March 26–29 1996. Kluwer Academic Publisher.
6. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), January 1985.
7. D. Gelernter. Multiple tuple spaces in Linda. In *Proceedings of PARLE*, volume 365 of *LNCS*, 1989.
8. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
9. T.J. LeBlanc, J.M. Mellor-Crummey, and R.J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.
10. D.M. Lyons and A.J. Hendriks. Planning for reactive robot behavior. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Nice, France, May 1992.
11. D.C. Marinescu, J.E. Lumpp, T.L. Casavant, and H.J. Siegel. Models for monitoring and debugging tools for parallel and distributed software. *Journal of Parallel and Distributed Computing*, 9:171–184, 1990.
12. A. Omicini, E. Denti, and A. Natali. Agent coordination and control through logic theories. In *Topics in Artificial Intelligence - 4th Congress of the Italian Association for Artificial Intelligence, AI*IA'95*, volume 992 of *LNAI*, pages 439–450, Firenze, Italy, October 11–13 1995. Springer-Verlag.
13. Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog Library*, 1994.

14. Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog User's Manual*, 1994.
15. F. Zanichelli, S. Caselli, A. Natali, and A. Omicini. A multi-agent framework and programming environment for autonomous robotics. In *Proceedings of the International Conference on Robotics and Automation (ICRA '94)*, pages 3501–3506, S. Diego, CA, USA, May 1994.