Expressive Power of the $\mathcal{ACLT}$ Reaction Specification Language

Enrico Denti   Antonio Natali   Andrea Omicini

LIA - *Laboratorio d'Informatica Avanzata*
Dipartimento di Elettronica, Informatica e Sistemistica
Università di Bologna
Viale Risorgimento, 2 - Bologna, Italy
{edenti, anatali, aomicini}@deis.unibo.it

Abstract. The notion of *programmable coordination medium* as a shared communication device whose behaviour can be defined according to the global system needs has indeed an impact over the design of multi-component software system. In fact, it allows coordination systems to be organised around a high-level inter-component interaction protocol, while the low-level details are delegated to properly programmed communication abstractions. This leads to a shift of (part of) the computational charge of a multi-component system from the communicating components (agents) to the communication device, and raises the issue of the expressive power of a programmable coordination medium.

As an example, this paper discusses the case of ReSpecT, the first-order logic language used by the $\mathcal{ACLT}$ coordination model to specify and define the behaviour of its multiple programmable logic tuple spaces. Its expressiveness is compared with that of Petri Nets (and of some of the variants and extensions), taken as a reference as one of the most used formalisms for the specification and design of concurrent and distributed systems. In particular, we show that ReSpecT is Turing-powerful. This makes it possible for the designer of a multi-component architecture to freely split the system's computational load between the communicating components and the communication device, so that local and global properties of a coordination system can be respectively embodied into the coordination entities ($\mathcal{ACLT}$ agents) and the coordination media ($\mathcal{ACLT}$ programmable logic tuple spaces) in the most natural way.

Keywords: *Multi-agent systems, Coordination, Tuple Spaces, Petri Nets, Programmable Coordination Media, Turing equivalence*

Contents

1 Introduction 3

2 Essential features of the $\mathcal{ACLT}$ model 4
- 2.1 The basics of $\mathcal{ACLT}$ reaction model . . . 5
- 2.2 Reactions as transactions . . . 6
- 2.3 From reactions to Programmable Coordination Media . . . 7

3 Mapping Petri Nets onto ReSpecT 8
- 3.1 An example . . . 9
- 3.2 The general mapping . . . 10

4 Mapping enhanced Petri Nets onto ReSpecT 11
- 4.1 Coloured Petri Nets and Predicate Petri Nets . . . 11
- 4.2 Petri Nets with Inhibition Arcs . . . 12
- 4.3 Priority Petri Nets . . . 14
- 4.4 Timed Petri Nets and Time Petri Nets . . . 15

5 Turing-equivalence 16
- 5.1 Turing-equivalence via Petri Nets . . . 16
- 5.2 Turing-equivalence via Register Machine . . . 18

6 Design of multi-component systems 19
- 6.1 Dining Philosophers . . . 19

7 Related works and conclusions 23

1 Introduction

Due to the increased complexity of software systems, a lot of work has been devoted in the last years to the study of coordination architectures [14, 6]. Many of the current researches concentrate on those communication mechanisms and abstractions enabling a coordination model to substantially decrease the complexity of the design and management of multi-component software systems. Blackboard-based coordination models like Linda [12] have shown how the typical limitations of point-to-point communication protocols can be in principle overcome, by exploiting *generative communication* as well as a shared communication abstraction.

In particular, Linda tuple spaces suggest that a key-role in the design and development of multi-component software systems should be played by the coordination media. In [7], we discussed the benefits of shifting computational properties from coordination entities to coordination media, suggesting the notion of *programmable coordination medium*. This notion is based on the full observability of all communication events, as well as on the programmability of the *reactions* to such events as a way to extend/determine the behaviour of the communication abstraction. This allows multi-component systems to be designed around the shared communication abstraction, whose behaviour can be tailored according to the system needs, while components can be designed following straightforward interaction protocols [10].

How much of a system behaviour should depend on the components, and, conversely, how much on the programmable communication device, is essentially a design problem, which cannot be solved in general, but only with respect to a particular application domain, according to given system goals. What has to be established in general is what computational power is needed and should be built in the coordination medium, and how expressive the language used to program its behaviour should be.

The $\mathcal{ACLT}$ [17] tuple space is an example of programmable coordination medium [7], that is, a shared communication abstraction whose behaviour can be defined by programming reactions to communication events through ReSpecT (*Re*action *Spec*ification *T*uples), the $\mathcal{ACLT}$'s logic tuple-based reaction specification language. Then, in this work, we will discuss the expressive power of ReSpecT as a meaningful example of programming language for a programmable communication abstraction. Based on the same (Linda-like) computational paradigm of the $\mathcal{ACLT}$ coordination model, ReSpecT allows the behaviour of an $\mathcal{ACLT}$ tuple space in response to an $\mathcal{ACLT}$ communication primitive to be specified through (meta-level) first-order logic tuples.

In order to discuss the expressive power of ReSpecT, we will compare it to the most used formalism for the description and specification of concurrent and distributed systems, Petri Nets, and to its most known variants and extensions. After presenting the essential features of the $\mathcal{ACLT}$ coordination model and of ReSpecT (Section 2), we will show how the behaviour of any Petri Net (Section 3), and of many of its extensions (Section 4) can be captured by a suitably programmed $\mathcal{ACLT}$ tuple space, by giving a general scheme for mapping any Petri Net onto a set of ReSpecT reactions. Section 5 will then show the Turing-equivalence of ReSpecT: first, indirectly, by mapping it onto Turing-powerful Petri Nets with Inhibition Arcs (Subsection 5.1), then, directly, by capturing the behaviour of the Shepherdson and Sturgis' register machine through a set of ReSpecT reactions (Subsection 5.2).

Finally, Section 6 discusses a simple example of a multi-agent system, by comparing the expressive power of Petri Nets and of ReSpecT from a design viewpoint. Section 7 contains final remarks and conclusions.

2 Essential features of the $\mathcal{ACLT}$ model

The $\mathcal{ACLT}$ coordination model, first presented in [17, 9] and later refined and discussed in [8, 10, 7], moves from the basic observation that, when combining components to build new applications, interaction constitutes a fundamental dimension [20] which can potentially make a multi-component system be something more than the simple sum of its parts. For this goal to become effective, a coordination model is required which provides the ability to express global system behaviour issues: typically, this requirement translates into an explicit coordination abstraction embedding some notion of globality - for instance, blackboards and tuple spaces.

However, the behaviour of such coordination abstractions is usually fixed once and for all, so it is quite hard to tailor it to the required overall system's behaviour. As a result, agents have to take care of the interaction protocols explicitly: they cannot abstract from the details of coordination policies, and are forcibly designed at a lower abstraction level than desirable. For the same reason, agent design often becomes unnecessarily complex.

The $\mathcal{ACLT}$ approach is based instead on the idea that an effective communication abstraction should be flexible and programmable [7], so as to be able to take most of the coordination tasks on itself instead of charging the application agents. To this purpose, $\mathcal{ACLT}$ adopts the notion of tuple space à la Linda [12] as its basic communication device, but enhances it by (*i*) raising the observability level from communication state (i.e., tuples) to communication events (i.e., operations on tuples), and by (*ii*) making it possible to define activities (called *reactions*) [8, 10] to be triggered upon the occurrence of some given communication event(s). In this way, the actual effect of the execution of a communication operation can be extended as needed, yet in a non-intrusive way (i.e., leaving the semantics of the basic communication primitive untouched). In addition, the reaction execution model makes application agents perceive all the effects of a single communication operation as a single-step transition of the state of the communication abstraction: so, at the agents' perception level, the behaviour of the coordination medium can be made as complex as desired, by suitably programming reactions [10, 7].

More generally, communication occurs in $\mathcal{ACLT}$ through a *multiplicity* [8, 10, 13] of *named*, *logic* tuple spaces [4, 5], which are collections of first-order unitary clauses, uniquely identified by a ground term. Logic tuple spaces can be interpreted either as simple communication channels, or as knowledge repositories: this latter reading suggests that they can be used as logic theories, where deductive activities can be performed over the current communication state. For this purpose, $\mathcal{ACLT}$ provides a set of *demo* primitives, as well as a coherent notion of logic consequence in a time-dependent environment [17, 9]. Since reactions are specified using the same communication pattern used for agent interaction (i.e. logic tuples and operations over tuple spaces), this feature may lead in perspective to systems able to reason about themselves, and possibly to self-modify their behaviour dynamically, according to the overall system goals.

2.1 The basics of $\mathcal{ACLT}$ reaction model

While $\mathcal{ACLT}$ (Linda-like) primitives have basically the same semantics as Linda ones, the behaviour of an $\mathcal{ACLT}$ tuple space can be extended by exploiting *reactions* to add new effects to communication events. More precisely, the $\mathcal{ACLT}$ reaction model is based on the idea of making communication events observable at the system level. For this purpose, any *physical* (communication) *event* can be associated with one or more *logical events*: then, *reactions* are defined so as to be triggered in response to the occurrence of such logical events.

Reactions are defined using the reaction specification language ReSpecT, which is based on first-order logic tuples. In ReSpecT, each logical event is denoted by a unique name, and associated to physical communication operations by a special tuple of the form `map(`*`OpName`*`,`*`Event`*`)`. So, each time an operation *`OpName`* is performed on the tuple space, a logical event *`Event`* conceptually occurs. The association between operations and events can be either one-to-one, one-to-many, many-to-one, or many-to-many: correspondingly, one `map(`*`OpName`*`,`*`Event`*`)` tuple is defined for each pair (*Operation*,*Event*) to be associated. So, multiple logical events can be connected to the same physical event, and multiple physical events can correspond to the same logical event.

Reactions are specified through tuples of the form `react(`*`Event`*`,`*`Body`*`)`, where *`Event`* is a logical event, and *`Body`* a *reaction body*. Whenever a logical event occurs, the corresponding reactions are triggered, executing their reaction bodies. Each reaction body is syntactically defined as a conjunction of *reaction goals*, which are either state primitives (`current_agent/1`, `current_op/1`, ...), term predicates (term equality/inequality, term unifiability/non-unifiability, ...), or operations over tuple spaces (`out_r`, `in_r`, `rd_r`, `no_r`)[1]. In the special but frequent case that only one single physical event is to be mapped onto a single logical event, the `map/2` clause is useless, and a simplified syntax can be used, expressing the reaction directly through a single `reaction(`*`Operation`*`,`*`Body`*`)` tuple.

Clearly, this shortcut syntax could be used as a general alternative to the `map/2` + `react/2` syntax defined before: however, uncoupling the mapping from the actual reaction specification is useful whenever the same reaction body has to be reused for many different physical operations, without having to rewrite it many times. This is the case, for instance, of the tracer presented in [8] and reported again below, which is built around a single reaction aimed to make the occurred communication event observable: by mapping all communication events onto a single logical event, the `map/2` + `react/2` syntax makes it possible to write the reaction body only once, instead of writing n `reaction/2` tuples sharing the same reaction body.

[1] `out_r` basically works as a conventional `out`, while `in_r` and `rd_r` correspond to `in_noblock` and `rd_noblock`, respectively. `no_r` is the dual operation of `rd_r`, succeeding when no matching tuple is found in the tuple space. Blocking primitives are not allowed inside reaction goals.

```
map(out,traceEvent).          react(traceEvent, ( current_op(Op),
map(in,traceEvent).               current_tuple(T), current_agent(Ag),
map(rd,traceEvent).               out_r(event(agent(Ag),
map(in_noblock,traceEvent).           performed(operation(Op),
map(rd_noblock,traceEvent).           on_tuple(T)))) )).
```

However, throughout this paper we will mainly exploit the simplest `reaction/2` syntax.

Reactions can be defined for all communication primitives, as well as for tuple operations performed by reactions themselves. Thus, *`OpName`* in a `map(`*`OpName`*`,`*`Event`*`)` tuple can be either an `out`, `in`, `rd`, `in_noblock`, `rd_noblock`, or an `out_r`, `in_r`, `rd_r` constant. Correspondingly, the term *`Operation`* in a `reaction(`*`Operation`*`,`*`Body`*`)` tuple can be either a `out(`T`)`, `in(`T`)`, `rd(`T`)`, `in_noblock(`T`)`, `rd_noblock(`T`)`, or an `out_r(`T`)`, `in_r(`T`)`, `rd_r(`T`)`, `no_r(`T`)` term, where T is a logic tuple.

Reactions are executed only after the corresponding communication event has actually occurred: so, in particular, when a reaction to an `out(`*`Tuple`*`)` primitive is triggered, the emitted *Tuple* is already in the tuple space.

Instead, `in` and `rd` communication primitives can be seen as made of two distinct communication events: the first query phase, when a tuple template is provided (the *pre* phase), and the subsequent answer phase, when a unifying tuple is eventually returned to the querying agent (the *post* phase).[2] According to that, $\mathcal{ACLT}$ allows different reactions to be associated to each of these two phases [8], by means of the `pre/0` and `post/0` predicates, which succeed only in the corresponding phase. So, for instance, when the reactions possibly associated to the *post* phase of an `in(`*`Tuple`*`)` primitive are executed, the returned tuple (i.e., the one unifying with *Tuple*) is no longer in the tuple space, while it is still there during the execution of reactions possibly associated to the *pre* phase of the primitive.

The same considerations basically apply to `in_noblock` and `rd_noblock`, too: since, however, these primitive may fail, $\mathcal{ACLT}$ allows different reactions to be executed in the *post* phase of a `in_noblock` or a `rd_noblock`, by means of the `success/0` and `failure/0` predicates, which, as intuitive, succeed only in the case that the current non-blocking primitive succeeded or failed, respectively.[3]

2.2 Reactions as transactions

We define a *successful reaction* as one whose reaction goals are all executed successfully; instead, if even one reaction goal fails, the reaction aborts. Only successful reactions produce effects, while failed reactions yield no result at all.

Since reaction goals are executed sequentially inside a reaction body, their relative order may influence the result of the reaction [10]. So, for instance, if there are no `p/1` tuples in the tuple space, and the logical event `event` occurs, the reaction

```
react(event, (in_r(p(X), out_r(q(X)), out_r(p(1)) ))
```

[2] Correspondingly, the `current_tuple` primitive refers to the tuple template in the query phase, and to the unified tuple in the answer phase.

[3] If the primitive failed, `current_tuple` refers to the tuple template both in the query and in the answer phase.

fails and produces no effect, while the reaction

```
react(event, (out_r(p(1)), in_r(p(X), out_r(q(X)) ))
```

succeeds, and eventually adds the tuple `q(1)` to the default tuple space.

At the system level, $\mathcal{ACLT}$ reactions are executed atomically with a transaction semantics: the (potentially multiple) effects of a successful reaction are carried out in a single transition of the tuple space state. Consequently, outside a reaction there is no way to perceive the multiplicity of effects possibly produced by the sequential execution of the reaction itself. So, in the case the reaction succeeds, all its side-effect operations are rendered simultaneously, leading to a single observable state transition. Instead, a failed reaction is virtually cancelled, as if it had never been executed, and yields no effect at all. Consider, for instance, the following reaction:

```
reaction(out(p(_)), (in_r(p(a)), in_r(p(X)), out_r(pp(a,X)) ))
```

Each time a new tuple `p/1` is inserted in the tuple space with an `out`, this reaction checks for the presence of two `p/1` tuples (whose one should be `p(a)`) and replaces them with one single `pp/2` tuple in case they are found. If some reaction goal fails (possibly because there is only one `p/1` tuple instead of the two required), no tuples are actually removed from the tuple space, nor are any other side-effects ever produced. If the reaction succeeds, instead, all its associated side-effects (removal of two tuples, and addition of a third) are realised altogether: in this case, the simultaneous presence of the two `p/1` tuples is perceived by the agents as a single `pp/2` tuple.

Since a reaction body can contain primitives like `out_r`, `in_r`, `rd_r`, and `no_r`, for which specific reactions could be defined, the successful completion of a reaction can cause further reactions to be triggered. In order to guarantee the transaction semantics, all these reactions are executed only *after* the triggering reaction has been successfully completed: so, no reaction nesting can take place.

2.3 From reactions to Programmable Coordination Media

Since multiple `react/2` tuples can be specified for a given logical event - as well as multiple `reaction/2` for the same communication primitive -, multiple reactions may be triggered at the same time. Moreover, as discussed in the previous Subsection, further reactions may be triggered as a consequence of the successful completion of another reaction, as a reaction body can contain operations over the tuple space in its turn.

According to the $\mathcal{ACLT}$ reaction execution model, reactions are executed as mutually-independent actions, in a non-deterministic order. However, it is guaranteed that all reactions executed as a consequence of an agent-triggered communication event (both triggered *directly* by the event and *indirectly* by other reactions produced by the event) are actually executed *before* serving any other agent-triggered communication event.

In this way, agents can only perceive the final result of the execution of the communication event and the set of all the reactions it triggered (directly and indirectly). For instance, if the following reaction has been defined,

```
reaction( out(p(s(X))), ( in_r(p(s(X))), out_r(p(X)) ))
```

an agent suspended on an `in(p(s(0)))` operation will not be waked up, as normally expected, by an `out(p(s(0)))` operation, as this reaction will first replace the `p(s(0))` tuple with a `p(0)` tuple. Then, only the `p(0)` tuple is visible at the agent's perception level, while the `p(s(0))` tuple is not perceived.

This behaviour introduces a new kind of tuple space state transition at the agent perception level, and enhances the expressive power of the coordination model. In fact, thanks to the $\mathcal{ACLT}$ reaction execution model described above, agents still perceive the response of a tuple space to a communication event as a single-step transition of the tuple space state, but this transition is no longer necessarily simple (adding/deleting one tuple) and fixed by the model, like in Linda, but can be made as complex as desired. From agents' perspective, for instance, the previous reaction

```
reaction( out(p(_)), ( in_r(p(a)), in_r(p(X)), out_r(pp(a,X)) ))
```

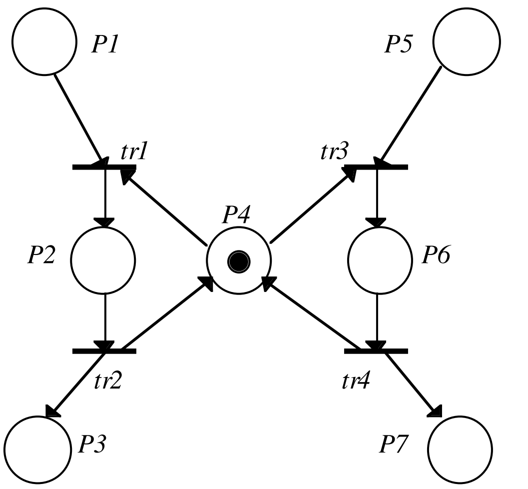
has the effect of making the simultaneous presence of the two `p/1` tuples unperceivable, and makes a single `out` operation result both in the removal of a tuple and in the insertion of another. In addition, the inserted tuple is not the one specified in the `out` operation, but is related both to that one and to the tuple space state.

3 Mapping Petri Nets onto ReSpecT

The basic idea is to exploit an $\mathcal{ACLT}$ tuple space to model a Petri Net. We will show how to define the behaviour of an $\mathcal{ACLT}$ tuple space so as to capture the behaviour of a Petri Net, by providing a suitable ReSpecT representation for all network items - namely, tokens, places, and transitions. Tokens constitute the Petri Net state information, so they are naturally represented by means of tuples. In particular, since the state of a Petri Net is represented by the number of tokens at each place, a tuple such as `place(`P`,`T`)` will indicate that T tokens are currently present at place P.

The behaviour of a Petri Net is defined by its transitions. Since they are inherently a synchronisation mechanism, they will be not represented relationally, but rather captured by proper reactions. So, there is no *direct* representation of the global network topology. Instead, the global state of the network is represented explicitly as a collection of `place/2` facts in the tuple space, along with a multiplicity of local representations of partial network structures, constituted by reactions. Each reaction models a single transition, and consistently manipulates the `place/2` tuples representing the tokens at the input and output places of the transition itself. More precisely, a reaction modelling a transition is triggered whenever the net's state changes, and performs three distinct conceptual steps: (*i*) retrieves the current knowledge about all the input places of the transition, (*ii*) checks the enabling conditions of the transition itself, and, in case they are verified, (*iii*) updates the net's state as required.

Given the transactional semantics of ReSpecT reactions, these steps are automatically performed with the required all-or-nothing semantics, in an atomic way. As a further result, only one transition is actually triggered at a time, even though multiple transitions may, in principle, be enabled. Thus, in the case that multiple transitions are enabled by a given set of tokens in a given set of places, only one will be (non-deterministically) selected




```
% INITIAL STATE
place(p1,0).  place(p2,0).  place(p3,0).
place(p4,s(0)).
place(p5,0).  place(p6,0).  place(p7,0).

% REACTIONS MAPPING TRANSITIONS
% transition tr1
reaction(out_r(place(_,_)), (
    in_r(place(p1,s(N1))), out_r(place(p1,N1)),
    in_r(place(p4,s(N4))), out_r(place(p4,N4)),
    in_r(place(p2,N2)), out_r(place(p2,s(N2)))
)).
% transition tr2
reaction(out_r(place(_,_)), (
    in_r(place(p2,s(N2))), out_r(place(p2,N2)),
    in_r(place(p4,N4)), out_r(place(p4,s(N4))),
    in_r(place(p3,N3)), out_r(place(p3,s(N3)))
)).
% transition tr3
reaction(out_r(place(_,_)), (
    in_r(place(p5,s(N5))), out_r(place(p5,N5)),
    in_r(place(p4,s(N4))), out_r(place(p4,N4)),
    in_r(place(p6,N6)), out_r(place(p6,s(N6)))
)).
% transition tr4
reaction(out_r(place(_,_)), (
    in_r(place(p6,s(N6))), out_r(place(p6,N6)),
    in_r(place(p4,N4)), out_r(place(p4,s(N4))),
    in_r(place(p7,N7)), out_r(place(p7,s(N7)))
)).
```

Figure 1. *The initial state of a Petri Net represented as* `place/2` *tuples, along with four reactions representing the four net's transitions*

for execution at a given time, and its execution will be performed before other reactions are considered. Other transitions will be either triggered or not, based on the evolved net state, as required.

3.1 An example

Before presenting the global mapping scheme, in order to help intuition, we will consider a very simple network such as the one shown in Figure 1, which represents two agents, A (*P1*/*P2*/*P3*, *tr1*, *tr2*) and B (*P5*/*P6*/*P7*, *tr3*, *tr4*), synchronising for a critical section through the semaphore *P4*.[4]

The initial state of the network is represented in the tuple space by seven `place/2` tuples, specifying that all places, except for *P4*, are initially empty. *P4*, instead, has one token. Transitions *tr1*–*tr4* are represented by four reactions, each one associated to a distinct transition, thus providing a local mapping of that part of the network structure. Due to the network symmetry, the code for *tr1* and *tr2* in Figure 1 is substantially identical to that for *tr3* and *tr4*, respectively.[5] Since transitions are triggered following net's state changes, which require new `place/2` tuples to be emitted to reflect the updated situation, triggering events are represented by `out_r(place(P,N))` operations. Token numbers and arc weights are represented through the `s/1` function (where `s(N)` is to be interpreted

[4] Of course, the activation of agent A (respectively, B) depends on the occurrence of a new token in place *P1* (respectively, *P5*); moreover, place *P3* (respectively, *P7*) may be connected to *P1* (respectively, *P5*) if cyclic activities are to be modelled.

[5] In fact, a more compact and elegant ReSpecT code could be easily defined for the network of Figure 1, by exploiting the network symmetry for place naming, and by properly using logic variables in the reaction bodies. However, this would be an "ad hoc" solution, which would not be relevant here, where the main issue is the general mapping between Petri Nets and ReSpecT.

as the successor of integer *N*), along with the constant `0` (representing, of course, the arithmetical zero). As usual, arcs with no explicit labelling are supposed to weight 1.

It is worth noting that, given the transactional semantics of $\mathcal{ACLT}$ reactions, if even one of the reaction subgoal fails, no side-effects are performed at all, as expected by an aborted transaction. In particular, this means that if one token is present in places *P1*, *P4* and *P5*, both *tr1* and *tr3* are enabled at the same time. Then, one of them is selected non-deterministically, conquer the critical section, and succeed, while the second one will be aborted due to the failure in trying to get the semaphore token at place *P4*.

In order to model the insertion of new tokens in one of the starting places (here, *P1* or *P5*), which should result in replacing the existing proper `place/2` tuple with a new one indicating the incremented amount of tokens at such a place, a new kind of event should be considered and properly handled. In particular, a tuple like `newtoken(P)` may be used to express that a new token has appeared[6] at place *P*, and a suitable reaction may be programmed (for instance, as shown in Subsection 3.2) to intercept `out(newtoken(P))` operations, and increment the token number at place *P* as appropriate. So, for instance, the addition of a new token at *P1* would be modelled by the communication event `out(newtoken(p1))`, resulting in the atomic substitution of the current `place(p1,N)` tuple with the new tuple `place(p1,s(N))`. Such a replacement, in its turn, will activate transition reactions, animating the network as expected in the case that one or more transitions are fired as a consequence of such an event.

3.2 The general mapping

This Section presents the general mapping rules which can be used to model a Petri Net with an $\mathcal{ACLT}$ tuple space using ReSpecT. The required steps can be outlined as follows:

1. Each place *P* is represented by a `place(P,N)` tuple, recording the number *N* of the tokens currently present at *P*. Obviously, the initial number of tokens at each place is given by the initial state of the Petri Net (represented by the initial state of the tuple space).

2. The insertion of a new token at place *P* is modelled by the emission of a tuple `newtoken(P)`. The following reaction transforms such an event into the atomic replacement of the involved `place/2` tuple:

```
reaction(out(newtoken(P)), (in_r(newtoken(P)),
    in_r(place(P,N)), out_r(place(P,s(N))) ))
```

3. Each transition is represented by a reaction catching any network state change - i.e., any `out_r(place(P,N))` operation. More precisely, each transition is represented by an ReSpecT tuple of the form `reaction(out_r(place(_,_), ReactionBody)`, whose reaction body contains a token substitution sequence for each input and output arc of the transition, encoding the required token removals and generations.

[6] If needed, this approach could be easily generalised to express the simultaneous insertion of N tokens at place P. However, since this is not essential to our purposes, we will leave this generalisation aside henceforth.

> A *token substitution sequence* for a place `P` is a sequence of two operations (one `in_r` followed by one `out_r`) replacing the current `place(P,N)` tuple with a new one, whose token number information is modified - i.e., incremented or decremented - as required by the modelled transition.

Consequently, whenever the net's state changes, all reactions associated to transitions are fired. Then, reactions corresponding to transitions which are still disabled will just abort, while reactions associated to newly-enabled transitions will start executing in a non-deterministic order, so that reactions' execution maps exactly transitions' competition. Moreover, since such a successful reaction modifies the token number at all output places of its associated transition, everything goes as if new tokens had been directly inserted at such places: that is, new reactions will be activated, animating the whole network in cascade, as expected, until leaf places - if any - are reached.

As an example of application of these rules, let us describe how the code of the previous example can easily be obtained given the Petri Net of Figure 1. First, since the net has seven places, whose only *P4* contains one token, the initial state consists of seven `place/2` tuples, all representing zero tokens except for the one regarding *P4*, which records the presence of one token as the `s(0)` term. Then, the four transitions are represented as four ReSpecT reactions, made of as many token substitution sequences as the number of arcs starting or ending at each transition. For instance, the reaction modelling transition *tr1* is made of three token substitution sequences, one for each arc starting or ending at *tr1*: the first one corresponding to the input arc coming from place *P1*, the second one for the input arc originating at *P4*, the third one for the output arc towards *P2*. In particular, the two input arcs (of weight 1) are represented as token substitution sequences where the token number is decremented by 1, while the output arc is captured by a token substitution sequence where such a number is incremented of the same amount.

4 Mapping enhanced Petri Nets onto ReSpecT

The above mapping captures the standard Petri Net definition. There are, however, several variants and extensions, which either improve the network readability without substantially affecting its expressive power in terms of computational properties, or, conversely, definitely have an impact on such an aspect, possibly raising the expressive power up to that of a Turing-equivalent formalism [3, 16].

In this Section, we will discuss how the $\mathcal{ACLT}$ reaction model may accommodate most of such variants, too; in particular, we will consider Coloured Petri Nets, Predicate Petri Nets, Priority Petri Nets, Petri Nets with Inhibition Arcs, and Time and Timed Petri Nets.

4.1 Coloured Petri Nets and Predicate Petri Nets

In ordinary Petri Nets, tokens are anonymous and indistinguishable one from each other. However, it is sometimes useful to be able to distinguish different kinds of tokens, representing different events of the real world, so as to operate differently according to the current token type. This is the case, for instance, of message-passing systems, where different actions are usually to be performed in response to (the reception of) different messages.

Coloured Petri Nets associate tokens to *colours*, so that at each place P there may be N_C tokens of colour C [3, 16]. Transition arcs are then labelled by either a set of colours, like $\langle green, blue \rangle$, or a variable like $\langle X \rangle$. In the first case, the corresponding input place must have at least one token (or in general W tokens, depending on the arc weights) of each listed colour (here, one *green* token and one *blue* token), while in the second case all transitions labelled with $\langle X \rangle$ must have one token of the *same* type - namely, one of the colours indicated in the domain of X. So, for instance, if X is given the domain $\{red, yellow\}$, all input places of a transition whose arcs are labelled $\langle X \rangle$ must either all have a *red* token, or, alternatively, all have a *yellow* token, for the transition to be possibly enabled.

Colours are quite easy to capture in the $\mathcal{ACLT}$ reaction model, once that the different token types are mapped onto correspondingly different symbols. This may be achieved by replacing the single `0` constant symbol with a set of *token symbols*, one for each colour. As a result, for instance, the tuple `place(p1,yellow)` indicates no *yellow* tokens at place `p1`, while the tuple `place(p2,s(s(red)))` means that two *red* tokens are present at place `p2`.

Transition behaviours are easy to capture, too, since each reaction has still simply to look for the tokens needed by each input arc, and to update input and output places consistently, now taking care of token symbols. In fact, to represent the case where a transition requires one token for each specified colour, what is needed is that proper token symbols are used inside `in_r` and `out_r` primitives. In the second case, where the constraint is that the same colour is used for all (input and output) arcs labelled with the same variable, the same logic variable is to be used as a token symbol for all arcs involved to obtain the required token matching.

Predicate Petri Nets [3, 16] are yet another variant, where each token may be associated to multiple colours. Since, however, their properties are basically the same as standard Coloured Petri Nets, we consider here the latter ones only.

Finally, a relevant issue concerns the computational properties of these extended networks. In fact, according to [3], while there is always an ordinary Petri Net equivalent to the coloured network provided if the set of possible colours is finite, such an equivalence no longer holds if the set of possible colours is infinite, in which case the extended formalism becomes Turing-equivalent. The fact that our approach provides a mapping where the set of possible colours in enumerably-infinite (because any logic term can be used instead of the constant `0`) suggests that the computational power of the $\mathcal{ACLT}$ Reaction Specification Language is Turing-equivalent, too. We do not stress this issue here, however, as it will be discussed more deeply in the following Subsection, with reference to extended Petri Nets with Inhibition Arcs, as well as in Section 5.

4.2 Petri Nets with Inhibition Arcs

A classical problem that ordinary Petri Nets cannot address is the so-called "zero-test", which consists of the inability to check for the absence of tokens at a given place. As a

```
% TRANSITION tr1 - STANDARD
reaction(out_r(place(_,_)), (
    in_r(place(p3,s(N3))), out_r(place(p3,N3)),
    in_r(place(p1,N1)), out_r(place(p1,s(N1)))
)).
```

```
% TRANSITION tr1 - WITH INHIBITION ARC
reaction(out_r(place(_,_)), ( rd_r(place(p2,0)),
    in_r(place(p3,s(N3))), out_r(place(p3,N3)),
    in_r(place(p1,N1)), out_r(place(p1,s(N1)))
)).
```

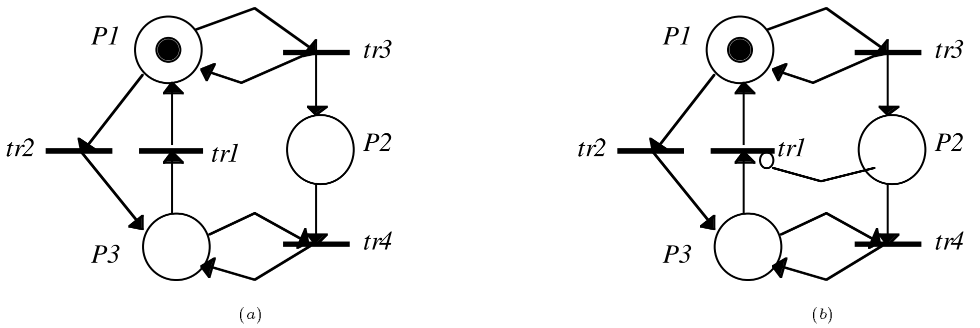


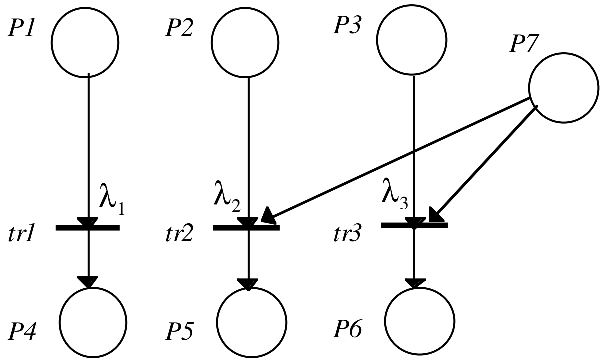
Figure 2. *The producer/consumer example in standard version* (a) *and with Inhibition Arc* (b)*, along with the corresponding* ReSpecT *code for transition tr1*

consequence, some concepts are impossible to model in the standard Petri Net formalism. For instance, in the simple two-agent system (a consumer and a producer) shown in Figure 2(a), it is impossible to constrain the producer to restart producing only after the consumer agent has used all objects (tokens) accumulated in the buffer place *P2*. Such impossibility is inherent to the problem, as standard Petri Nets provide no explicit representation for the "zero token" concept.

According to [3], a partial solution to this problem is possible in ordinary Petri Nets only if the network is *limited* (i.e., there is an upper bound to the number of tokens which may be present in all places at any time), in which case a special technique (called *complementary places technique*) may be applied. In the example of Figure 2, however, the network is clearly unlimited, as there is no upper bound to the number of tokens (objects) which may be produced and accumulated at place *P2*.

In order to achieve such expressive power in the general case, the formalism needs to be enriched. *Inhibition Arcs* [3, 16] provide this enhanced (Turing-equivalent) expressive power, by enabling a transition only if the connected place is empty.

Interestingly enough, this enhancement is immediately captured by ReSpecT with virtually no change or addition at all to the mapping scheme presented in Subsection 3.2. In fact, in order to bound a transition to be enabled only in case there are no tokens at a given place P, the corresponding reaction has simply to include an explicit `rd_r(place(`P`,0))` operation. As an example, Figure 2(b) reports the variant of the producer/consumer example with an inhibition arc, as well as the corresponding ReSpecT code, where the `rd_r(place(p2,0))` operation has been added to the reaction body. It is worth noting that this feature directly depends on the explicit representation for the zero-token state as a `place(`P`,0)` tuple, which is precisely what ordinary Petri Nets are unable to express: actually, knowing that knowledge is zero is inherently different - and more expressive - than having no knowledge at all. As a further consequence, no special mechanism is needed in order to capture the event of the "last token removal", yielding to a zero-token condition.




```
% TRANSITION tr1
reaction(out_r(place(_,_)), (
    in_r(place(p1,s(N1))), out_r(place(p1,N1)),
    in_r(place(p4,N4)), out_r(place(p4,s(N4)))
)).

% TRANSITION tr2
reaction(out_r(place(_,_)), (
    rd_r(place(p1,0)),
    in_r(place(p2,s(N2))), out_r(place(p2,N2)),
    in_r(place(p7,s(N7))), out_r(place(p7,N7)),
    in_r(place(p5,N5)), out_r(place(p5,s(N5)))
)).

% TRANSITION tr3
reaction(out_r(place(_,_)), (
    rd_r(place(p1,0)), rd_r(place(p2,0)),
    in_r(place(p3,s(N3))), out_r(place(p3,N3)),
    in_r(place(p7,s(N7))), out_r(place(p7,N7)),
    in_r(place(p6,N6)), out_r(place(p6,s(N6)))
)).
```

Figure 3. *A Priority Petri Net with priorities* $\lambda_1 > \lambda_2 > \lambda_3$*, along with the corresponding* ReSpecT *code*

For instance, in case both places *P2* and *P3* contain one token (thus keeping transition *tr1* disabled), when the token is removed from *P2*, transition *tr1* will be immediately triggered, due to the `place(p2,0)` tuple's insertion in the tuple space.

4.3 Priority Petri Nets

When multiple transitions are enabled, the ordinary Petri Net scheme requires that the one to be triggered is chosen non-deterministically. However, in many practical systems it is useful to introduce a *priority* relation between transitions, so that, when many of them are enabled, the triggered one is known a-priori. Such networks are called Priority Petri Nets [3], and their computational power is known to be Turing-equivalent. Actually, there is a direct correspondence between these networks and Petri Nets with Inhibition Arcs, as it is always possible to transform one kind of network in the other [3]. Intuitively, this is possible because saying that transition *tr1* has a higher priority than *tr2* means that *tr2* can be enabled only if *tr1* is disabled, which can be expressed by an inhibition arc (starting from *P1*) constraining *tr2* to be enabled only when *tr1* is not. In other words, priorities represent implicitly the same exclusion constraints that inhibition arcs represent explicitly.

For these reasons, the priority concept is simple to be captured by the $\mathcal{ACLT}$ reaction model, and the resulting ReSpecT code is very similar the one discussed in the previous Subsection for Petri Nets with Inhibition Arcs. More precisely, priorities can be expressed by embedding more knowledge in reactions than is usually done: reactions should (contain operations meant to) test also the input places of higher-priority transitions, instead of their input places only. In practice, this means that each reaction should check that both its own enabling conditions hold, and the enabling conditions required by its higher-priority transitions do not hold. As a result, knowledge implicitly embedded in priority relationships is expressed explicitly as a set of tests (i.e., `rd_r` primitives) on the condition of places enabling higher-priority transitions.

Figure 3 shows an example of how a Priority Petri Net can be mapped onto the $\mathcal{ACLT}$ reaction model according to the approach outlined above. There, λ_1, λ_2 and λ_3 denote priority of transitions *tr1*, *tr2* and *tr3*, respectively. Given the priority relationship $\lambda_1 > \lambda_2 > \lambda_3$, transition *tr1* is unconstrained, while transition *tr2* is constrained to be enabled only in case *tr1* is not, and *tr3* can be enabled only in the case that *tr2* - and, therefore, *tr1* too - cannot be enabled in their turn. Correspondingly, the reactions representing *tr1* can be written just following the general mapping rules, while the reactions mapping *tr2* must contain a test to check that *tr1* is disabled, and the reactions mapping *tr3* must contain a test to check that *both tr1* and *tr2* are disabled. The first condition translates as checking that *P1* contains no tokens, for which task a `rd_r(place(p1,0))` primitive can be used: interestingly, this corresponds to drawing an inhibition arc from *P1* to *tr2*. The second condition, instead, which must ensure that both *tr1* and *tr2* are disabled in order to let *tr3* be enabled, can be expressed by the conjunction `rd_r(place(p1,0)),rd_r(place(p2,0))`: once again, this corresponds to drawing a pair of inhibition arcs from *P1* and *P2* to *tr3*.

It is worth noting that even though *tr2* has two input places, only one has to be checked by the *tr3* reaction, since one of them (*P7*) is shared with *tr3*. In the case, instead, that *tr2* has another input place (say, *P8*), shared with no other transition, another reaction would have been needed to model *tr3*, checking *P1* and *P8*, so as to test enabling conditions of *tr2* disjunctively, as required. More generally, by considering for the sake of simplicity only arcs of weight 1, a transition like *tr2*, with *n2* input arcs, conditioned by a higher-priority transition like *tr1*, with *n1* input arcs, requires at most *n1* distinct reactions to be properly mapped onto the $\mathcal{ACLT}$ reaction model. Analogously, a transition like *tr3*, conditioned by transitions *tr1* and *tr2*, requires at most $n2 * n1$ distinct ReSpecT reactions. In case reactions share some input places, the check of the enabling conditions can be simplified, and fewer reactions are needed. In fact, in case that *tr3* shares $s1$ input places with *tr1* and $s2$ with *tr2*, it would exactly need $(n2 - s2) * (n1 - s1)$ reactions to be modelled. In the example of Figure 3, for instance, *tr3*, which shares 1 input place with *tr2*, and no one with *tr1*, requires $(2-1)*1 = 1$ reaction. In the case that *P8* is added, *tr3* would require instead $(3-1)*1 = 2$ reactions.

4.4 Timed Petri Nets and Time Petri Nets

Timed Petri Nets and Time Petri Nets are extensions of the ordinary Petri Net formalism aimed to explicitly include the time dimension in the model. More precisely, Timed Petri Nets associate a duration time to each transition, so that the "transition triggered" event is no longer instantaneous, but lasts exactly the specified time. Time Petri Nets, instead, specify two time instants for each transition: the first represents the minimum time that the transition must be enabled in order to be triggered, while the second one represents a deadline, at which the transition must necessarily be triggered.

In both cases, however, time is to be explicitly taken into account in transition handling. For this reason, such extensions cannot be captured by the $\mathcal{ACLT}$ reaction model, where no real-time notion is provided, and no guarantees are given about the absolute time at which a reaction will be eventually executed.

5 Turing-equivalence

Although computability theory usually refers to the Turing Machine as the abstract computer model, other abstract machines have been proposed as a reference model in literature during the years, such as the Minsky machine, or the register machine by Shepherdson and Sturgis [19]: however, they were all eventually recognised to be Turing-equivalent. So, any of these may be exploited to prove the Turing-equivalence of another formalism: here, the Shepherdson and Sturgis' machine will be used.

Basically, such a machine is defined as a triple $\langle R, L, S \rangle$, where R is a finite set of registers used to store arbitrarily large integer numbers, L is a set of labels used to denote statements, and S is a set of statements, representing the machine's program. Of course, the computational properties of this machine directly depend on which statements are enclosed in S.

Shepherdson and Sturgis proved [19] that such a machine is Turing-equivalent if S contains the statements $\mathtt{inc}(R_i)$, which increments the value of register R_i, $\mathtt{dec}(R_i)$, which decrements the value of register R_i provided that it is greater than zero, and $\mathtt{jmp}(R_i, \mathit{label})$, which jumps to the statement denoted by *label* if the content of register R_i is zero. The *jump* statement obviously implies that control flow normally selects the "following" statement, thus implicitly introducing a notion of program as a sequence of (labelled) statements. So, we will consider labelled statements of the form:

$$\mathit{label} : \mathtt{statement}\,,\ \mathit{next\_label}$$

where *label* identifies the `statement` whose execution has to be followed by that of the statement denoted by *next_label*.

Even more, [15] showed that the two following forms of statements suffices to represent any recursive function

a) $\mathit{label} : \mathtt{inc}(R_i),\ \mathit{next\_label}$

b) $\mathit{label} : [\text{if } R_i = 0]\ \mathtt{jmp}(R_i, \mathit{new\_label}),\ [\text{else}]\ \mathtt{dec}(R_i),\ \mathit{next\_label}$

Thus, the ability to capture these two instruction schemes is enough to prove the Turing-equivalence of any abstract machine.

5.1 Turing-equivalence via Petri Nets

A very simple way to show the Turing-equivalence of the $\mathcal{ACLT}$ Reaction Specification Language is by exploiting Turing-equivalence of Petri Nets with Inhibition Arcs, and the correspondence between such a formalism and ReSpecT, as discussed in Subsection 4.2. In particular, we will first remind [3, 18] how any Shepherdson and Sturgis' machine can be represented as a Petri Net with Inhibition Arc, and show then how each of the Petri Nets needed to model a register machine can be represented in terms of ReSpecT tuples, according to the scheme of Subsection 4.2.

For any register machine $\langle R, L, S \rangle$, then, we will define:

- a place PR_i for each register $R_i \in R$

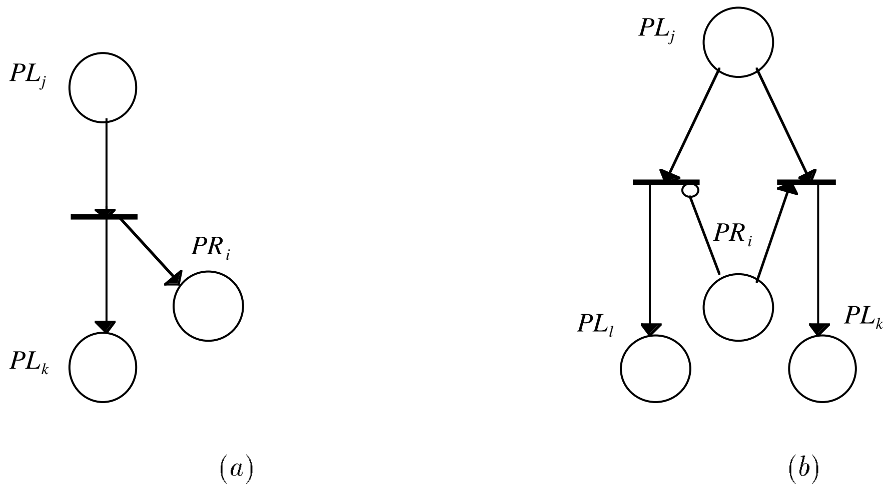


Figure 4. *Basic Petri Nets for computational equivalence with the register machine by Shepherdson and Sturgis*

- a place PL_j for each label $L_j \in L$
- an equivalent network for each statement $s \in S$. If $R_i \in R, L_j, and L_k, L_l \in L$, s will have one of the following forms:

  a) L_j: `inc(`R_i`)`, L_k

  b) L_j: [if $R_i = 0$] `jmp(`R_i, L_l`)`, [else] `dec(`R_i`)`,L_k

The two statement forms given above can be represented as Petri Nets as shown in Figure 4. Network (*a*) and (*b*) in Figure 4 can be simply represented in ReSpecT through the following reactions:

a)
```
reaction(out_r(place(_,_)), (
    in_r(place(pLj,s(NLj))), out_r(place(pLj,NLj)),
    in_r(place(pRi,NRi)), out_r(place(pRi,s(NRi))),
    in_r(place(pLk,NLk)), out_r(place(pLk,s(NLk)))
)).
```

b)
```
reaction(out_r(place(_,_)), (
    rd_r(place(pRi,0)),
    in_r(place(pLj,s(NLj))), out_r(place(pLj,NLj)),
    in_r(place(pLl,NLl)), out_r(place(pLl,s(NLl)))
)).
reaction(out_r(place(_,_)), (
    in_r(place(pRi,s(NRi))), out_r(place(pRi,NRi)),
    in_r(place(pLj,s(NLj))), out_r(place(pLj,NLj)),
    in_r(place(pLk,NLk)), out_r(place(pLk,s(NLk)))
)).
```

As detailed in [3, 18], given an initial state, the computation of a register machine exactly matches the triggering sequence of the corresponding extended Petri Net, and the content of each register at any time exactly equals the token number at the corresponding place. Analogously, an $\mathcal{ACLT}$ tuple space programmed as detailed above will trigger one ReSpecT reaction for any transition of the equivalent Petri Net, such that at any time of the computation the tuple space representation of the token numbers will exactly match the Petri Net's state. Thus, since Petri Nets with Inhibition Arcs are as powerful as Shepherdson and Sturgis' machine, so is ReSpecT.

5.2 Turing-equivalence via Register Machine

A very simple mapping is also possible, modelling a register machine as an $\mathcal{ACLT}$ tuple space, showing again the Turing-equivalent computational power of ReSpecT. For any register machine $\langle R, L, S\rangle$, then, we will define:

- a ground term `reg(`*`Ri`*`)` for every register $R_i \in R$.
- a ground term `label(`*`Lj`*`)` for every label $L_j \in L$.
- a tuple `value(reg(`*`Ri`*`),`*`Ni`*`)` for every register $R_i \in R$, where *`Ni`* is the integer value of R_i represented as a `s(s(...s(0)...))` term.
- a tuple `statement(label(`*`Lj`*`),`*`Statement`*`,label(`*`Lk`*`))` for every statement $s \in S$, where *`Statement`* has one of the following forms:

  (a) `inc(`*`reg(Ri)`*`)`

  (b) `jmpdec(`*`reg(Ri),label(Ll)`*`)`

- an *execution reaction* for each of the possible instructions to be executed:

(a)
```
reaction(out_r(exec(Label)), ( in_r(exec(Label))
     rd_r(statement(Label,inc(Register),NextLabel)),
     in_r(value(Register,N)), out_r(value(Register,s(N))),
     out_r(exec(NextLabel))
)).
```

(b_{jmp})
```
reaction(out_r(exec(Label)), ( in_r(exec(Label))
     rd_r(statement(Label,jmpdec(Register,JumpLabel),_)),
     rd_r(value(Register,0)),
     out_r(exec(JumpLabel))
)).
```

(b_{dec})
```
reaction(out_r(exec(Label)), ( in_r(exec(Label))
     rd_r(statement(Label,jmpdec(Register,_),NextLabel)),
     in_r(value(Register,s(N))), out_r(value(Register,N)),
     out_r(exec(NextLabel))
)).
```

- a *start reaction* of the form

```
reaction(out(start(Label)), out_r(exec(Label)) ).
```

A computation starts following an `out(start(`*StartLabel*`))` operation. Any emission of an `exec(`*Label*`)` tuple triggers the execution of the instruction labelled *Label*. In the case that it corresponds to an `inc` statement, reactions (b_{jmp}) and (b_{dec}) would abort, due to the `rd_r` test. Reaction (a) would succeed, instead, by substituting the tuple representing the value of the register involved with a new one whose value is incremented by one, and by making the computation continue through the emission of a new `exec/1` tuple. In case instead of a `jmpdec` instruction, reaction (a) immediately aborts, again due to the failure of the `rd_r` test. Then, only one among reactions (b_{jmp}) and (b_{dec}) succeeds, depending instead on the value of the register involved, as required by the instruction intended semantics. In the case that the involved register is zero (b_{jmp}), execution simply jumps somewhere according to the content of the `jmpdec` term. Otherwise (b_{dec}), the register value is decremented by one, and execution continues with the emission of a new `exec/1` tuple.

6 Design of multi-component systems

The design of a multi-component software system deeply depends on the expressiveness of the language as well as on the abstractions provided by the model used for its definition and specification. Thus, in order to measure the expressive power of ReSpecT, it seems useful to briefly discuss its impact on the design of coordination systems. For this purpose, this Section will take a simple case of multi-agent system (the Dining Philosopher one), and discuss the different results of specifying it via Petri Nets, or alternatively via ReSpecT. Given the computational equivalence of this two Turing-powerful formalisms, this comparison should help putting in evidence some of the benefits of the $\mathcal{ACLT}$ reaction model in terms of the metaphors it provides, the easiness of the design, and the flexibility of the resulting systems.

6.1 Dining Philosophers

In this classical Dining Philosophers problem [11], agents (philosophers) may either be thinking or eating. They need a pair of forks in order to start eating, but unfortunately forks are shared resources. From our viewpoint, two characteristics are fundamental:

- philosophers show a deliberative activity, i.e. they decide autonomously when they want to start eating and stop thinking, or viceversa;
- to avoid deadlock situations, a philosopher must either get both forks he needs to eat or get none, i.e. fork acquisition must be performed through a transaction. Moreover, to ensure fairness, fork release should better be atomic.

At a first glance, a Petri Net-based solution could be designed directly, first introducing proper places and tokens to represent available resources (forks), then building a network around them so as to capture the required agent behaviour. In the very simple case of a two-philosopher system, such a Petri Net could be like the one in Figure 5: its mapping

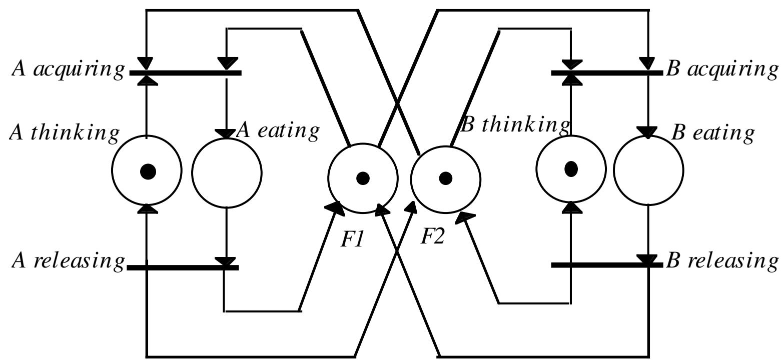


Figure 5. *A simple two-philosopher system*

onto $\mathcal{ACLT}$ reactions, given the rules presented in Sections 3 and 4, is straightforward.

This network does not represent agents' deliberative activity at all: in fact, places and transitions are thought to represent agent and resource (fork) state, and not its intentions, which could be anyway represented explicitly as tokens, by adding two places (i.e., *want_to_eat*, and *want_to_think*) for every agent. In fact, being the result of a bottom-up design, this network describes interaction at the lowest possible level - that of physical resources to be handled - making it very hard to understand how the agent interaction protocol actually works. Moreover, as far as flexibility is concerned, the generalisation to $N > 2$ agents is quite straightforward, but makes the network very complex, even for such a simple problem. In addition, more complex extensions or variants to the problem [10, 7] would result in a nearly complete re-design of the corresponding Petri Net specification.

Designing at such a low level - although possible - is unnecessary in ReSpecT, and a higher abstraction level can be adopted. The target machine of ReSpecT (that is, $\mathcal{ACLT}$ programmable logic tuple space) makes it possible to define first the high-level agent interaction protocol, then the behaviour of the communication abstraction, thus tailoring it according to higher design needs. In particular, a designer could abstract from the actual handling of physical resources, delegating it to (the reactions programmed onto) the programmable coordination abstraction. As a result, ReSpecT allows the agent interaction protocol to be designed to be as straightforward as desired, and to be built into a stand alone interaction component (a suitably programmed tuple space). Such a

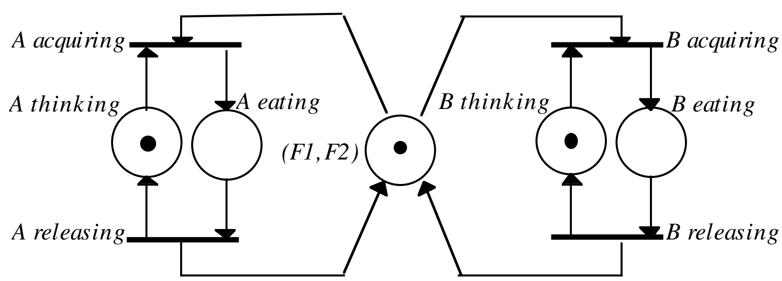


Figure 6. *The two-philosopher system from a philosopher viewpoint*

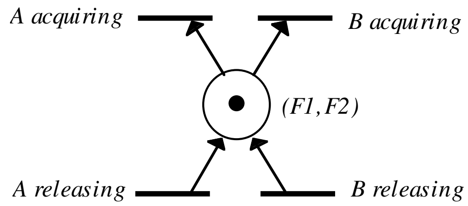


Figure 7. *The core of the interaction protocol at the agent perception level*

component is then in charge of the lower-level resource handling policy, whose design can then be uncoupled from the agent design, and whose details are unperceivable at the agent level.

Adopting this approach, the solution to the Dining Philosopher problem comes out differently. In fact, from the agents' viewpoint, a philosopher would like just to *acquire* the two forks he needs to eat when hungry, and to *release* them when satiated: all the synchronisation issues related to transactionally get and atomically release such forks are (or should be) just a matter of details to be delegated to the underlying machinery. So, the $\mathcal{ACLT}$ reaction model naturally suggests to re-think the network of Figure 5, and re-formulate it as shown in Figure 6, where pairs of resources (fork pairs) are handled atomically. The task of filling the gap between the abstraction level of Figure 6 (based on the abstract concept of pairs of resources) and the physical level (where single forks are actually available) is delegated to the (properly programmed) target machine.

Since what is relevant to interaction is agent intention, which is manifested by *acquire/release* macro-events, rather than agent internal state, Figure 7 shows the "core" of the network, obtained from that of Figure 6 by suppressing the modelling of single agent states. This core can be actually thought as representing the interaction protocol *at the agent perception level*: it is up to the programmable coordination medium to translate such high-level intentions into the proper operations on actual resources (forks). At the resource level, in fact, no fork pairs, but only single forks are actually available. Therefore, the required abstract concept of fork pair must be synthesised by properly synchronising fork acquisition and release. So, the same behaviour specified at the agent level by the Petri Net of Figure 7 is expressed at resource level by the network of Figure 8. It is worth noting that in order to be able to implement such a bridge between higher and lower perception levels, requiring higher-level macro-events to be expressed as lower-level micro-events, atomicity and transactionality are essential features. That is why such a

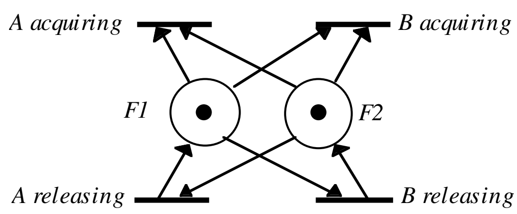


Figure 8. *The core of the interaction protocol at the resource perception level*

mapping can be easily reported onto $\mathcal{ACLT}$ reaction scheme, where these properties are built-in in the model. As a result, the code solving the Dining Philosopher problem can be expressed in $\mathcal{ACLT}$ so that

- *at the agent perception level:*
  - fork pairs are represented by means of the `forks/2` tuple (so, in the case of our two-philosopher system, agent A acquires and releases `forks(1,2)`, while agent B uses `forks(2,1)`);
  - coherently, fork acquisition is modelled by an `in(forks(`*F1*`,`*F2*`))` operation (possibly suspensive), while fork release is modelled by an `out(forks(`*F1*`,`*F2*`))` operation;
- *at the resource perception level:*
  - single available forks are represented in the tuple space by means of the `fork/1` tuple (so, in the case of our two-philosopher system, `fork(1)` and `fork(2)` are initially present in the tuple space);
  - the tuple space is programmed to bridge between single forks and fork pairs, handling `fork/1` and `forks/2` tuples consistently.

The ReSpecT reaction code (part of it, below) implementing this bridging may then look like the following:

```
reaction( out(forks(1,2)), ( in_r(forks(1,2)),
    out_r(fork(1)), out_r(fork(2)) )).
reaction( out(forks(2,1)), ( in_r(forks(2,1)),
    out_r(fork(1)), out_r(fork(2)) )).
reaction( in(forks(1,2)), ( pre, out_r(required(1,2)) )).
reaction( in(forks(2,1)), ( pre, out_r(required(2,1)) )).
reaction( in(forks(1,2)), ( post, in_r(required(1,2)) )).
reaction( in(forks(2,1)), ( post, in_r(required(2,1)) )).
...
```

Although correct and working, the above ReSpecT code shows unnecessary duplications, which are directly related to the Petri Net structure - that is, in practice, to the expressiveness of the Petri Net formalism. In fact, the reactions implementing fork acquisition and release by agent A and agent B are basically identical, except for the forks involved - (1,2) in the first case, and (2,1) in the second. Actually, the network topology shows clear symmetries between its left and right sides. Things would become even more evident in an N-philosopher system ($N > 2$), which would have N nearly identical network sections, leading to N nearly identical reactions - one for each fork pair to be considered.

While such a duplication cannot be avoided in a standard Petri Net, it can be avoided in the ReSpecT code, by suitably exploiting logic variables. This corresponds to implementing the elementary network "template" shown in Figure 9. As shown in [7] (where it is fully explained), the complete ReSpecT code for N dining philosophers is the following:

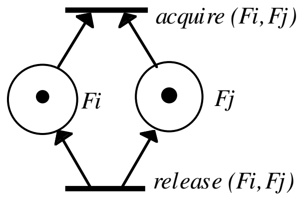


Figure 9. *The template Petri Net for fork handling at the resource level*

```
reaction( out(forks(F1,F2)), ( in_r(forks(F1,F2)),
    out_r(fork(F1)), out_r(fork(F2)) )).
reaction( in(forks(F1,F2)), ( pre, out_r(required(F1,F2)) )).
reaction( in(forks(F1,F2)), ( post, in_r(required(F1,F2)) )).
reaction( out_r(required(F1,F2)), (
    in_r(fork(F1)), in_r(fork(F2)), out_r(forks(F1,F2)) )).
reaction( out_r(fork(F)), ( rd_r(required(F1,F)),
    in_r(fork(F1)), in_r(fork(F)), out_r(forks(F1,F)) )).
reaction( out_r(fork(F)), ( rd_r(required(F,F2)),
    in_r(fork(F)), in_r(fork(F2)), out_r(forks(F,F2)) )).
```

It should be obvious that, while this does not enhance the computational power, it does definitely have an impact over the expressiveness of formalism used for the system specification.

Obviously, all the networks shown in this Section can be mapped onto the $\mathcal{ACLT}$ reaction scheme, from the first one of Figure 5 to the last ones. However, this example should have shown that the architecture suggested by the ReSpecT model is rather different with respect to the one shown in Figure 5, which results from the Petri Net approach. In particular, the ReSpecT approach naturally leads to split the coordination protocol in two cooperating layers (in the case above, the high-level *acquire/release* philosopher protocol, and the low-level forks' handling), since it inherently supports two different perception levels: the agent level (that of the actual interaction) and the system level (that of ReSpecT reactions). This leads to the corresponding separation of different level tasks between different components - agents for the high-level part, and the supporting communication abstraction for the lower-level part - which seems indeed a relevant feature for an effective design of coordination systems.

7 Related works and conclusions

The ReSpecT computational model discussed here is based on the notion of *reaction*, like other known coordination models. However, some remarkable differences are worth to be put in evidence.

In Gamma [1], for instance, very general coordination laws are specified in terms of *reaction conditions* and consequent *actions*. However, the model provides neither a communication abstraction, nor any agent interaction protocol. Even more, the model does not account for agent deliberative activity, as it can be argued from the Dining

Philosopher example shown in [2], where reactions are the only means for the evolution of a multi-agent system based on Gamma.

More similar to $\mathcal{ACLT}$ is the ESP coordination language [4, 5], which exploits reactiveness of the tuple space. There, agents are modelled in terms of guarded computations which are triggered in reaction to given states of the tuple space. Thus, reactiveness in ESP is still linked to the state of the communication, while $\mathcal{ACLT}$ allows to specify reactions to communication events. Moreover, it should be noted that the computational shift from the coordination entities to the communication abstraction is even stronger than in $\mathcal{ACLT}$, as ESP reduces the notion of agent to a purely reactive execution thread, and does not account for any agent deliberative activity.

According to our perception, the design and development of coordination architectures may actually take advantage from being based on programmable coordination media, that is communication abstractions whose behaviour is not fixed, but can be programmed so as to accomplish the overall system goals. As a remarkable example of programmable coordination medium, the $\mathcal{ACLT}$ coordination model provides logic tuple spaces whose behaviour can be programmed by defining reactions to communication events through ReSpecT, the $\mathcal{ACLT}$ reaction specification language. Since ReSpecT is Turing-powerful, as shown in this paper, the computational load of an $\mathcal{ACLT}$-based multi-component system can be freely splitted between the communicating components and the communication device according to the application needs. In particular, $\mathcal{ACLT}$-coordinated systems can be designed such that their local and global properties can be embodied into the coordination entities ($\mathcal{ACLT}$ agents) and the coordination media ($\mathcal{ACLT}$programmable logic tuple spaces), respectively, in the most natural way. The resulting system are easy to design and flexible: as shown in [7], for instance, new interaction policies for agent cooperation/competition are simple to be implemented and built in the communication abstraction with minimal or no modifications at all to the agents' behaviours.

References

[1] J.-P. Banâtre and D. Le Métayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, November 1990.

[2] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[3] G.W. Brahms. *Petri Nets: theory and practice.* Masson (italian edition), 1985.

[4] A. Brogi and P. Ciancarini. The concurrent language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1), January 1991.

[5] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12, 1994.

[6] P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28(2), June 1996.

[7] E. Denti, A. Natali, and A. Omicini. Programmable coordination media, 1997. Second International Conference, COORDINATION'97, Berlin, Germany, September 1–3, 1997.

[8] E. Denti, A. Natali, A. Omicini, and M. Venuti. An extensible framework for the development of coordinated applications, 1996. First International Conference, COORDINATION'96, Cesena, Italy, April 15–17, 1996.

[9] E. Denti, A. Natali, A. Omicini, and M. Venuti. Logic tuple spaces for the coordination of heterogeneous agents. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems*, pages 147–160. Kluwer Academic Publishers, 1996. First International Workshop "Frontiers of Combining Systems", FroCoS'96, Munich, Germany, March 26-29, 1996.

[10] E. Denti and A. Omicini. Designing multi-agent systems around an extensible communication abstraction. In A. Cesta and P.-Y. Schobbens, editors, *Proceedings of the 4th ModelAge Workshop on Formal Models of Agents, Certosa di Pontignano, Italy, January 15–18, 1997*, pages 87–97. National Research Council of Italy, 1997. To be published by Springer-Verlag in the LNAI Series.

[11] E.W. Dijkstra. *Co-operating sequential processes*. Academic Press, London, 1965.

[12] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), January 1985.

[13] D. Gelernter. Multiple tuple spaces in Linda. In *Proceedings of PARLE*, volume 365 of *LNCS*, 1989.

[14] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[15] M.L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Englewood Cliff, 1967.

[16] T. Murata. Petri Nets: properties, analysis, and applications. In *Proceedings of the IEEE*, volume 77(4), 1989.

[17] 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.

[18] J.L. Peterson. *Petri Net theory and the modelling of systems*. Prentice-Hall, Englewood Cliffs (NJ), 1981.

[19] J.C. Shepherdson and H.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.

[20] P. Wegner. Interactive foundations of computing. Technical report, Brown University, Providence (RI), August 1996.