Mapping REO on TuCSoN/ReSpecT

- Distributed System Project -

Federica Pecci	federica.pecci2@studio.unibo.it
Margherita Pecorelli	margherita.pecorelli@studio.unibo.it
Chiara Varini	chiara.varini@studio.unibo.it

Alma Mater Studiorum - Università di Bologna
Via Sacchi 3 – 47521 Cesena, Italia

Abstract

Nowadays modern systems are becoming very modular and made of independent components, which can be implemented using different languages (e.g. Java and C-like languages). Since these components are different from each other and considering that every component interface depends on its context, it can occur that a gap is produced among these interfaces in a distributed system. For this purpose, the focus has been moved from system design to system coordination and interaction. As a consequence, new formalisms, a.k.a. *coordination models*, have been developed. A coordination model is a conceptual framework for modelling the space of interaction among components in multi-component system. One of the models for the coordination of a agent-based distributed system is the tuple-based one.

A particular implementation of this paradigm is the *tuple center*. As reported in [1], a tuple center is a tuple space whose behaviour can be defined by means of reactions to communication events. More precisely, a tuple centre is a communication abstraction which is perceived by the interacting entities as a standard tuple space, but whose behaviour, in response to communication events, can be defined so as to embed the laws of coordination.

Besides TuCSoN/ReSpecT, REO is another exogenous coordination model based on channel calculus. REO can be used as a flexible *glue code* to implement connectors that handle system components. In addition, it defines the primitives which allow to compose channels: the communication medium that provides a temporal/spatial decoupling between two components.

Index

1 TuCSoN/ReSpecT

1.1 Tuple-based Coordination with TuCSoN

TuCSoN is a tuple-based model for coordination of distributed process. A TuCSoN system is a collection of agents and tuple centers working together in a (possibly) distributed set of nodes [4]. TuCSoN agents (coordinables) communicate, synchronise and cooperate through *tuple centers* (coordination media) by storing, reading and consuming tuples and behaviours in an associative way. TuCSoN tuple center is a *hybrid* communication media that combines two different approaches: data-driven and control-driven.
Data-driven model is used to handle shared informations. This model makes interaction protocol simple yet expressive. The main limits of this model are:

- there is not distinction between perceived information and its representation;
- the fixed behaviour of coordination medium;
- the policies of coordination cannot be embedded into the coordination medium.

Control-driven model is used to overcome the above limits in that way:

- keeping clearly separate information representation and perception;
- enriching the behaviour of the coordination medium;
- allowing behaviours to be no longer fixed;
- embedding coordination laws into coordination medium.

TuCSoN was conceived with this model so that it can handle internet applications based on network-aware and mobile agents [2].

The main features provided by Tucson are:

- the TuCSoN coordination space can be used either as a global space or a local one. The agents can access resocurces, services or other tuple centers trought the absolute name (tc@node?operation(tuple)) or the local name(tc?operation(tuple)). These names are referred to tuple centers located on a specific network node. In particular, the semantic used is the same of the file system one [2];
- the TuCSoN coordination media can be used to specify behaviours such as embedded coordination laws in coordination medium. Different communication abstraction can encapsulate different coordination laws, providing system designers with a finer granularity for the implementation of global coordination policy [2]. In order to define a new behaviour, it is possible to specify computational activities called reactions via a Reaction Specification Language (for instance ReSpecT, as it will be explained below).

A reaction is a set of non-blocking operations, and as a success/failure transactional semantic, moreover, it can freely access and modify the tuple center state and also get all the informations related to the triggering communication event. By this way it becomes possible to design more complex, and not fixed operations. This makes it possible to uncouple the agent's view of the tuple centre from the tuple centre actual state.

1.2 Programming Interaction with ReSpecT

ReSpecT is a suitable language to use in combination with the TuCSoN model and is a behaviour specification language and its acronym stands for Reaction Specification Tuples. This language allows to easily implement behaviours and manage tuple centers [1], which are basically tuple spaces but enriched with behaviours.

1.2.1 Main Features

Firstly, ReSpecT has a twofold nature [3]:

- **procedural**: it is possible to define a series of computations which happens inside the tuple center and are called reactions as well;
- **declarative**: it is offered the possibility to link events to some reactions inside the tuple center.

Secondly, how reported in [2], a tuple center contains tuples which express the declarative nature of the language and can be divided into two main categories:

- ordinary tuples, whose collection constitutes its tuple space part;
- *specification tuples*, whose collection constitutes its *specification space* part. The template for this type of tuple is `reaction(E,G,R)`, thus, every time an event (`E`) is triggered all the reactions (`R`) associated to it are non-deterministically executed. More in details, given a ReSpecT event Ev, a specification tuple `reaction(E,G,R)` associates a reaction `R`Θ to `Ev` if and only if Θ `= mgu(E,Ev)` – where mgu is the most general unifier, as defined in logic programming – and guard predicate `G` is true.

Additionally, input parameters of a specification tuple `reaction(E,G,R)` are [3]:

- **event descriptor `E`**, that is an event and it is used to match with admissible events;
- **guard `G`**, which could be true or false and reactions associated to E will be executed only if the guard is true;

- **sequence `R`**, that is a reaction body, which consists of some reaction goals. A reaction goal is either a primitive invocation, a predicate recovering properties of the event, or some logic-based computation. Sequences of reaction goals are executed transactionally with an overall success / failure semantics

Moreover, it is reported that ReSpecT is characterised by the presence of the following two couples of predicates [2]:

- **`pre/0` and `post/0` predicates**, so that one may define reactions succeeding only in the pre or post phase;
- **`success/0` and `failure/0`**: which succeed only in the case that the current non-blocking primitive succeeded or failed, respectively.

To conclude, this language handles very well side effects, deleting them even if one or more reactions (associated to an event) fail and bringing the tuple center back to a solid state.

1.2.2 Dining Philosophers Example

Since ReSpecT is used in a distributed context and a common problem affecting distributed systems is deadlock, it could be interesting analysing Dining Philosophers problem, which is the traditional one chosen to exemplify solutions to the deadlock situation. This problem regards how to handle philosophers' access, through their chopsticks, to a spaghetti bowl which is on a table, considering that each philosopher spends his time only doing two activities: thinking and eating. As a consequence, in ReSpecT, assuming that N philosophers are distributed along the network, it is possible to do the following analogies:

- **tuple centre `seat(i,j)`**: each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`, which means that the associated philosopher needs chopstick pair `chops(i,j)` in order to eat;
- **tuple `chop(i)`**: each chopstick i is represented as a tuple `chop(i)` in the table tuple centre;
- **tuple `wanna_eat/wanna_think`**: each philosopher expresses his intention to eat/think by emitting a tuple `wanna_eat/wanna_think` in his `seat(i,j)` tuple centre.

Everything else is handled automatically in ReSpecT and this because it is embedded in the tuple centre behaviour. In addiction, a distinction must be drawn between N individual tuple centres (`seat(i,j)`) and 1 social tuple centre (`table`), in fact, each philosopher can perform either *Individual Interaction* or *Social Interaction*. These two types of interaction can be described as follow:

1. Individual Interaction concerns Philosopher–seat interaction (use) and has the following main features:

- **tuple `philosopher()`**: four states are represented by this tuple, in particular, they are: thinking, waiting to eat, eating, waiting to think. These states are determined by i) the `out(wanna_eat)`/`out(wanna_think)` invocations/actions, which express the philosopher's intentions ii) the interaction with the table tuple centre, expressing the availability of chop resources;
- **tuple `chops(i,j)`**: this tuple only occurs in tuple centre `seat(i,j)` in the philosopher (eating) state.

State transitions only occur when they are safe i) from waiting to think to thinking only when chopsticks are safely back on the table, ii) from waiting to eat to eating only when chopsticks are actually at the seat. Thus, before passing from one state to another chopsticks have to be in a well-defined tuple center (that is seat or table one).

ReSpecT code for `seat(i,j)` tuple centres can be found in Listing 1.

```
reaction( out(wanna_eat), (operation, invocation), (
    in(philosopher(thinking)),
        out(philosopher(waiting_to_eat)),
    current_target(seat(C1,C2)), table@node ?
        in(chops(C1,C2)) )).
reaction( out(wanna_eat), (operation, completion),
    in(wanna_eat)).
reaction( in(chops(C1,C2)), (link_out, completion),
    in(philosopher(waiting_to_eat)), out(philosopher(eating)),
    out(chops(C1,C2)) )).
reaction( out(wanna_think), (operation, invocation), (
    in(philosopher(eating)),
        out(philosopher(waiting_to_think)),
    current_target(seat(C1,C2)), in(chops(C1,C2)),
    table@node ? out(chops(C1,C2)) )).
reaction( out(wanna_think), (operation, completion),
    in(wanna_think) ).
 reaction( out(chops(C1,C2)), (link_out, completion), (
    in(philosopher(waiting_to_think)),
        out(philosopher(thinking)) )).
```

Listing 1: ReSpecT code for `seat(i,j)` tuple centres

2. Social Interaction concerns Seat-table interaction (link) and has the following main features:

   - **tuple centre `seat(i,j)`**: it requires tuple `chops(i,j)` from table tuple centre or it returns tuple`chops(i,j)` to table tuple centre;
   - **tuple `chops(i,j)`**: tuple centre table transforms tuple `chops(i,j)` into a tuple pair `chop(i)`, `chop(j)` whenever required, and back `chop(i)`, `chop(j)` into `chops(i,j)` whenever required and possible.

ReSpecT code for table tuple centre can be found in Listing 2.

```
reaction( out(chops(C1,C2)), (link_in, completion), (
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (link_in, invocation), (
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (link_in, completion), (
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Listing 2: ReSpecT code for table tuple centre

What most characterises Distributed Dining Philosophers example in ReSpecT is:

- **full separation of concerns**: i) philosophers just express their intentions, in terms of simple tuples ii) individual tuple centres - `seat(i,j)` - handle individual behaviours and state, and mediate interaction of individuals with the social tuple centre - table iii) the social tuple centre - `table` - deals with shared resources (`chop` tuples) and ensures global system properties (i.e. fairness and deadlock avoidance);

- **representation of the current distributed state**: it is consistent at any time if one looks at the coordination media. In fact, the representation is properly distributed, suitably encapsulated, accessible and represented in a declarative way.

1.3 TuCSoN Formalisation

A tuple centre is a tuple space with a behaviour specification $\sigma \in S$ where S is the reaction specification language adopted by the coordination model. A behaviour specification $\sigma \in S$ associates each communication event $e \in E$ to a (possible empty) multi-set of reactions of R, where R is the set of all the admissible reactions according to the reaction specification language S. If σ associates event $e \in E$ to reaction $r \in R$, we say that the occurrence of e in a tuple centre whose behaviour specification is σ triggers r, which is then executed by the tuple centre. With respect to a tuple space, then, a tuple centre also contains triggered reactions in the form of pairs $(e, r) \in E \times R = Z$, recording that reaction r triggered by event e is currently waiting to be executed. So, the state of a tuple centre can be expressed as a triple $\langle T, W, Z \rangle$, where

- T $\subset T$ is the set of the tuple descriptors currently in the tuple centre;
- W $\subset E$ is the set of the pending queries, that is, the agent-triggered requests for tuples accepted by the tuple centre, and waiting to be served;
- Z $\subset Z = E \times R$ is the set of the triggered reactions waiting to be executed.

Whilst each coordination model exploiting tuple centres relies on a specic reaction specification language, the abstract notion of tuple centre does not depend on the particular language adopted. So, in this context we abstract from the reaction specification language S by factorizing its role in terms of two functions, whose definition fully accounts for the semantics of S:

- the reaction specification function $Z_\sigma : E \to 2^Z$, where $\sigma \in S$
- the reaction evaluation function $E : Z \times 2^T \times 2^E \times 2^Z \to 2^T \times 2^E \times 2^Z \times 2^E$.

In particular, the reaction specification function Z puts communication events and reactions in relation according to a behaviour specification. In fact, Z takes a communication event $e \in E$ and maps it onto a set $Z_\sigma(e) \subset Z$ of triggered reactions, depending on the tuple centre behaviour specification σ. In particular, since the behaviour of a tuple centre defaults to the behaviour of a tuple space when $\sigma = \emptyset$, Z is always dened such that $\forall e \in E, Z(e) = \emptyset$. In turn, the reaction evaluation function E encapsulates the effects of executing reactions. In fact, E takes a triggered reaction $(e, r) \in Z$ and the state of a tuple centre $(T \subset T, W \subset E, Z \subset Z)$, and returns the new tuple centre state (T', W', Z') as well as a (possibly empty) set of resulting output communication events OE$\subset E$. The operational behaviour of a tuple centre can be modelled in terms of a transition system by (i) extending tuple space transitions listening (r1) and speaking (r2) so as to make them handle reaction specification, and by (ii) introducing a new reacting transition $(\to r)$ in charge of executing triggered reactions. Thus, given a behaviour specification σ and a function Z, a listening transition for a tuple centre extends transition rule (r1) as follows:

$$\frac{\text{if } W_p{=}W_s{=}\emptyset}{\langle T,W,\emptyset\rangle \xrightarrow{ie}_1 \langle T',W',Z_\sigma(ie)\rangle}, \qquad \text{[r3]}$$

where T' and W' depend in general on the intended effects of the incoming communication event $ie \in E$ over T and W , and $Z_\sigma(ie)$ is the set of the reactions triggered by ie according to σ. With respect to transition rule (r1), it is worth noting that a listening transition for a tuple centre is enabled only when Z=$\emptyset$, that is, when besides being no pending queries to be served there are no triggered reactions to be executed, too. Analogously, a speaking transition for a tuple centre extends (r2) as follows:

$$\frac{\text{if } w \in W_s \cup W_p}{\langle T,w \cup W,\emptyset\rangle \xrightarrow{o}_{e_s} \langle T',W',Z_\sigma(oe)\rangle}, \qquad \text{[r4]}$$

where $oe \in E$ is the communication event emitted by the tuple centre, both oe and T' result from serving pending query w, and $Z_\sigma(oe)$ is the set of the descriptors of the reactions triggered by oe according to behaviour specification σ. Again, with respect to transition rule (r2), it is worth noting that a speaking transition for a tuple centre is enabled only when there are no triggered reactions to be executed ($Z = \emptyset$). Whereas transition rules (r3) and (r4) extend (r1) and (r2) by handling reactions through the reaction specification function Z, a third reacting transition must be introduced to handle the execution of the triggered reactions based on the reaction evaluation function E. A reacting transition has the following general form:

$$\frac{if\ z \in Z}{\langle T,W,Z\rangle \xrightarrow{OE}_r \langle T',W',Z'\rangle}, \qquad \text{[r5]}$$

where $E(z,T,W,Z) = (T',W',Z',OE)$ denotes the result of the execution of triggered reaction z. According to (r5) precondition, no reacting transition is performed in absence of triggered reactions, as obvious. Along with the definition of Z and transition rule (r3) and (r4), this ensures that the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given ($\sigma = \emptyset$).

It should be noted that a tuple centre neither receives external events (i.e., listening transitions are not enabled) nor serves its pending queries (i.e., speaking transitions are not enabled) until it has executed all the triggered reactions (i.e., all admissible reacting transitions have been performed). In other words, all the reactions triggered by a communication event according to a tuple centre behaviour specification are executed by the tuple centre before handling any further request from agents. As a consequence, agents perceive the result of any communication event and the effects of its triggered reactions altogether as a whole, that is, as a single transition of the tuple centre state. This is precisely what makes it possible to program a tuple centre so as to exhibit a new observational behaviour, by expressing coordination rules in terms of a reaction

specification language, and embedding them into the coordination medium.

2 REO

REO, as reported in [5], is an exogenous coordination model based on channel calculus and exploits primitive channel types to build more complex *connectors*. It can be used as a language for coordination of concurrent processes or as a "glue language" for compositional construction of connectors that orchestrate component instances in a component-based system.

Channel-based communication model can easily reproduce the primitives of other communication models like message passing, shared spaces or remote procedure calls.

REO provides a set of primitive channel types with a well-defined behaviour that can be combined by the user in order to build his own connector topologies.

REO only defines how components connect and coordinate with each other, abstracting away from their implementation details. The component instances of a system can communicate outside only with input/output operations that they perform on a (dynamic) set of *channel ends*, which are connected to it.

A channel is the primitive point-to-point medium of communication that provides the basic temporal and spatial decoupling of the parties involved in a communication. It has not a fixed direction, but each channel has exactly two directed ends namely *Source* and *Sink*. The former accepts data into its channel, whereas, the latter offers data out of its channel.

Both components and connectors (composed by channels) are assumed to be mobile, that is when a component instance moves from one location to another the topology of channel connections is preserved: the channel ends connected to the component instance remain connected.

REO makes a distinction between the channel (channel end) operations and the node operations. The set of primitive operations on channels and channel ends is summarized in Table 1, on the other hand, the one performed on node can be found in Table 2.

It is important to notice that in Table 1, the names of all the operations begin with an underscore because they are to be used internally by REO only: (the active entities inside) component instances are not allowed to perform these operations directly.

Operation	Condition	Description
`_create(chantype)`	-	This operation performs `create(chantype)`, creates a node for each of the two resulting channel ends, and returns the identifiers of the two channel ends.
`_forget(e)`	N	Changes `e` such that it no longer refers to the channel end it designates.
`_move(e, loc)`	Y	Moves `e` to the location `loc`.

_connect([t,] e)	N	Connects the specified channel end, `e`, to the component instance that contains the active entity that performs this operation.
_disconnect(e)	N	Disconnects the specified channel end from the component instance that contains the active entity that performs this operation.
_wait([t,] conds)	N	Suspends the active entity that performs this operation, waiting for the conditions specified in `conds` to become true for the specified channel ends.
_read([t,] inp[, v[, pat]])	Y	Suspends the active entity that performs this operation, waiting for a value that can match with `pat`, to become available for reading from the sink channel end `inp` into the variable `v`. The read operation is non-destructive: the value is copied from the channel into the variable, but the original remains intact.
_take([t,] inp[, v[, pat]])	Y	This is the destructive variant of read: the channel loses the value that is read.
_write([t,] outp, v)	Y	Suspends the active entity that performs this operation, until it succeeds to write the value of the variable `v` to the source channel end `outp`.

Table 1: Primitive channel operations.

A connector is a set of channel ends and their connecting channels organized in a graph of nodes and edges such that:

- every channel end coincides on exactly one node;
- every node can have one or more channel ends on it
- if there is a channel between two nodes (not necessarily distinct), then they are connected by an edge.

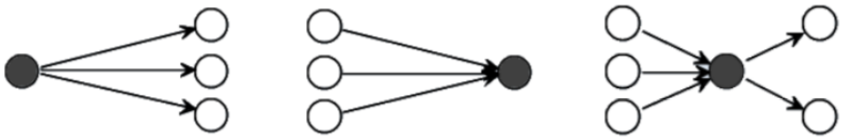
Given a channel end `x`, which is on a node `N`, the annotation $\hat{x}$ coincides with `N` and `[N]` is the set containing all the channel ends that belong to`N`.

It exists three different types of nodes as reported in [6]:

1. Source Node if $Src(N) \neq \emptyset \wedge Snk(N) = \emptyset$;

2. Sink Node if $Src(N) = \emptyset \wedge Snk(N) \neq \emptyset$;

3. Mixed Node if $Src(N) \neq \emptyset \wedge Snk(N) \neq \emptyset$.



(a) *Source Node.* (b) *Sink Node.* (c) *Mix Node.*

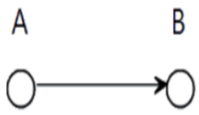
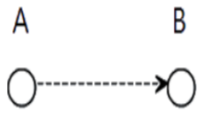
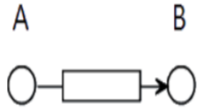
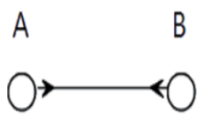
Each component can only link with Source and Sink nodes, whereas, Mixed nodes can be used within connectors.

Operation	Condition	Description
`create(chantype)`	-	This operation performs `create(chantype)`,creates a node for each of the two resulting channel ends, and returns the identifiers of the two channel ends.
`forget(e)`	N	This operation atomically performs the set of operations `forget(x)`, $\forall x \in [\hat{e}]$.
`move(e, loc)`	Y	This operation atomically performs the set of operations `move(x, loc)`, $\forall x \in [\hat{e}]$.
`connect([t,] e)`	N	If be is not a mixed node, this operation atomically performs the set of operations `connect([t,] x)`, $\forall x \in [\hat{e}]$.
`disconnect(e)`	N	This operation atomically performs the set of operations `disconnect(x)`, $\forall x \in [\hat{e}]$.
`wait([t,] nconds)`	N	This operation succeeds when the conditions specified in `nconds` become true.
`read([t,] e[, v[, pat]])`	Y	If be is a sink node connected to the component instance, this operation succeeds when a value compatible with `pat` is non-destructively read from any one of the channel ends `x` $\in [\hat{e}]$ into the variable `v`.
`take([t,] e[, v[, pat]])`	Y	If be is a sink node connected to the component instance, this operation succeeds when a value compatible with pat is taken from any one of the channel ends `x` $\in [\hat{e}]$ and read into the variable `v`.
`write([t,] e, v)`	Y	If be is a source node connected to the component instance, this operation succeeds when a copy of the value `v` is written to every channel end `x` $\in [\hat{e}]$ atomically.

join(e1, e2)	Y	If at least one of the nodes $\hat{e_1}$ and $\hat{e_2}$ is connected to the component instance, this operation merges them into a new node (i.e., after the join, $\hat{e_1}$ and $\hat{e_2}$ become synonyms for the same node).
split(e[, quoin])	N	This operation produces a new node `N` and splits the set of channel ends in $[\hat{e}]$ between the old $\hat{e}$ and `N`, according to the set of edges specified in `quoin`.
hide(e)	N	This operation hides the node be such that it cannot be modified in any other operation.

Table 2: Node Operations

Each connector is composed by simpler channels. REO provides some primitive channel types with a well-define behaviour summarised in Table 3 as reported in [6].

	A **Sync** channel has a source (A) and a sink end (B) and no buffer. It accepts a data item through its source end (A) if and only if it can synchronously dispense it through its sink (B).
	A **LossySync** channel is similar to a Sync channel except that it always accepts all data items through its source end (A). This channel loses the data item only whenever it cannot synchronously dispense it through its sink (B).
	A **FIFO1** channel represents an asynchronous channel with a buffer of capacity 1: it can contain at most one data item.
	A **SyncDrain** channel is a synchronous drain that accepts a data item through one of its ends if and only if it synchronously accepts a data item through its other end as well.

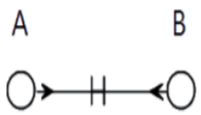
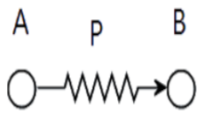
	An **AsyncDrain** channel is an asynchronous drain that accepts data items through exactly one of its source ends at a time and loses that data item.
	A **Filter(P)** channel is similar to a Sync channel except that it loses data items that do not match pattern P.

Table 3: Primitive channel types in REO

The generic behaviour of a channel c whose channel ends are $x_c = Src$ and $y_c = Sink$ is defined with the following two functions:

1. offers(y_c, p) is the multi-set of pairs $\langle y_c, d \rangle$ for each d in the multi-set of values that may be assigned to a variable v in a $take(0, y_c, v, p)$.

2. accepts(x_c, d) is true for a data item d if the state of c allows $write(0, x_c, d)$ to succeed.

2.1 REO Formalisation

REO is based on *Timed Data Streams*, a pair, $\langle \alpha, a \rangle$ of time (a) and data (α) streams with the interpretation that $\forall i \geq 0$, the data item $\alpha(i)$ appears at its corresponding time moment `a(i)`. Timed data streams, whose acronym is TDS, are used to model the flows of data through channel ends. A channel itself is just a (binary) relation between the two timed data streams associated with its two ends. A more complex connector is simply an n-ary relation among n timed data streams.

In order to designate whether or not the operation O is pending (on its respective node), the predicate Π(O) is used. Each channel behaviour is defined by two functions: offers and accepts, that can be formalised as follows:

$$\texttt{accepts}(N,p) = \begin{cases} \bigwedge_{p:\Pi(take(T,N,v,p))} d \ni p & \text{if } N \text{ is a sink node} \\ \bigwedge_{x \in Src(N)} accepts(x,d) & \text{otherwise} \end{cases}$$

$$\texttt{offers}(N,d) = \begin{cases} \biguplus_{d:\Pi(write(T,N,v,p))} \{\langle \epsilon, d \rangle | d \ni p\} & \text{if } N \text{ is a source node} \\ \biguplus_{x \in Sink(N)} offers(x,p) & \text{otherwise} \end{cases}$$

The syntax above is used to express the relationship between data (d) and pattern (p) and has the following meaning:

- $d \ni p$: d matches with p;
- $d \not\ni p$: d matches with p.

The primitive channel type formalisation is reported below.

2.1.1 Sync

$accepts(x_c, d) = accepts(\hat{y_c}, d)$

$offers(y_c, p) = \{\langle y_c, d \rangle | \langle z, d \rangle \in offers(\hat{x_c}, p)\}$

The channel Sync is formally defined as the relation:

$\langle \alpha, a \rangle Sync \langle \beta, b \rangle \equiv \alpha = \beta \wedge a = b$

The equation $\alpha = \beta$ states that every data item that goes into a Sync channel comes out in the exact same order. The equation a = b states that the arrival and the departure times of each data item are the same: there is no buffer in the channel for a data item to linger on for any length of time.

2.1.2 LossySync

$accepts(x_c, d) = true$

$offers(y_c, p) = \{\langle y_c, d\rangle | \langle z, d\rangle \in offers(\hat{x}_c, p)\}$

A LossySync channel is defined as the relation:

$$\langle \alpha, a\rangle LossySync\langle \beta, b\rangle \equiv \begin{cases} \alpha = \beta & \text{if } a = b \\ \langle \beta, b\rangle = \langle\rangle & otherwise \end{cases}$$

2.1.3 FIFO1

$accepts(x_c, d) = |B(c)| \leq 1$ where 1 is maximum buffer capacity

$$offers(y_c, p) = \begin{cases} \{\langle y_{c,B_1}\rangle\} & \text{if } B(c) \neq \langle\rangle \wedge B_1 \ni p \\ \emptyset & otherwise \end{cases}$$

A FIFO1 channel is defined as the relation:

$\langle \alpha, a\rangle FIFO1\langle \beta, b\rangle \equiv \alpha = \beta \wedge a < b < a'$

2.1.4 SyncDrain

In this channel x_c = Source channel-end and y_c = Source channel-end

(accept) $accepts(x_c, d1) = accepts(y_c, d2)$

offers $offers(y_c, p) = false$

A SyncDrain channel is defined as the relation:

$\langle \alpha, a\rangle SyncDrain\langle \beta, b\rangle \equiv a = b$

2.1.5 AsyncDrain

In this channel x_c = Source channel-end and y_c = Source channel-end

(accept) $accepts(x_c, d1) = not(accepts(y_c, d2))$

offers $offers(y_c, p) = false$

An AsyncDrain channel is defined as the relation:

$\langle \alpha, a\rangle AsyncDrain\langle \beta, b\rangle \equiv a \neq b$

2.1.6 Filter(P)

$accepts(x_c, d) = d \not\ni pat \vee accpets(\hat{y}_c, d)$

$offers(y_c, p) = \{\langle y_c, d\rangle \| \langle z, d\rangle \in offers(\hat{x}_c, p) \wedge d \ni pat\}$

A Filter(P) channel is defined as the relation:

$\langle \alpha, a\rangle Filter(pat)\langle \beta, b\rangle \equiv \alpha \ni pat \wedge \alpha = \beta \wedge a = b$

3 REO on TuCSoN/Respect

The goal of this chapter is to perform a mapping between the abstractions offered by Reo (peer to peer model) and those offered by TuCSoN/ReSpecT (tuple center model).

It has been chosen to map the main concepts of REO in abstractions provided by TuCSoN/ReSpecT in the following way:

- **component/coordinable** → agent;
- **node** → tuple center;
- **channel** → tuple center;
- **connector** → tuple center set.

In particular, in this project, each tuple center is identified by all the other tuples and agents, present in the connector, through its own IP address. Therefore,the IP address becomes the only discriminating factor with which it is possible to distinguish one component (node or channel) from another. This choice proved to be particularly useful and intuitive in a distributed context in which the code of each node/channel can be executed on a different machine and, thus, with a different IP. However, it would have been possible to identify the tuple centers also with other parameters, for example with full names expressed as a combination of the tuple center name with the address and a port. In this way it would be easy to create connectors and test them on a single machine, creating tuple centers that would use different ports. As regards TuCSoN/ReSpecT implementations, it will be presented, at first those of the 3 types of REO nodes (Source, Sink and Mixed), then those of the 6 basic channels of REO (Sync, LossySync, FIFO 1, Filter(P), SynchDrain e AsynchDrain). Each node/channel is associated with a set of reactions that, if configured on a tuple center, allow to coordinate the agents participating in the communication. In order to construct more complex connectors, it has been assumed the existence of an algorithm that takes a Reo connector as input and returns a set of TuCSoN/ReSpecT reactions as output, having an equivalent behaviour. In particular, a series of reactions (to be set on the different tuple centers) will be returned, remembering that there is a tuple center for each node/channel of the REO connector.

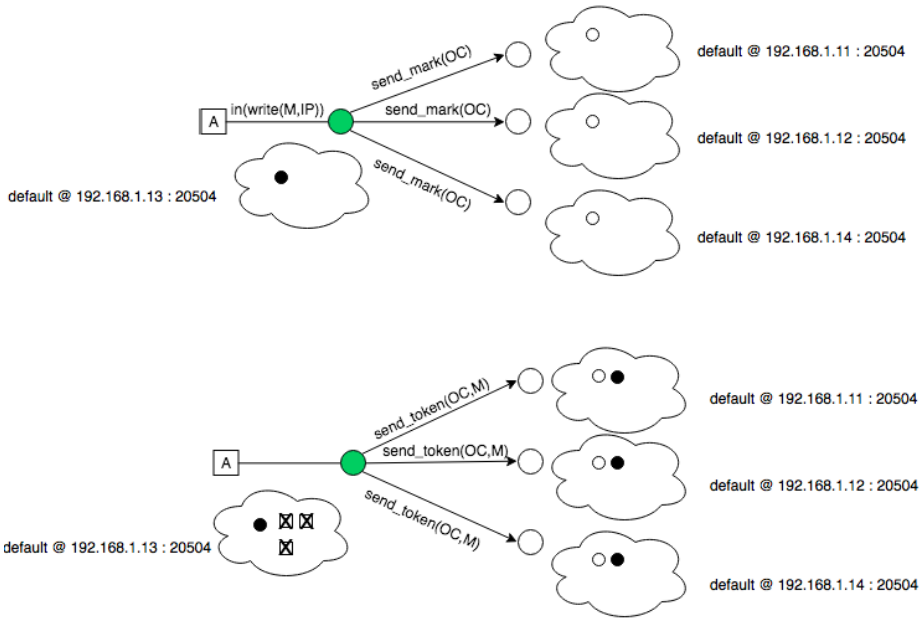
3.1 Nodes

3.1.1 Source Node

The behaviour of Source node is characterised by a sequence of operations that can be summarised in 3 steps:

- **sending availability to write**: when the agent connected to the node sends a write operation (*write(`M`, IP)*, where `M` represents a message and IP the address of the Source node), the node notifies its outgoing channels

Figure 1: Source node semantics



(*outgoing_ channels (L)*, where L is the list of channel IPs) that it is ready to write. The notification to the channels is done by sending a mark (*white_ circle*);

- **sending blue_circle**: when a Source node is connected to one or more SyncDrain channels (which are necessary to synchronise multiple channels), the node must notify the channel its write availability when the number of white squares is equal to the number of outgoing channels minus the number of SyncDrain channels connected. The notification is sent by sending a *blue_ circle*.
- **message sending**: the actual message is sent once the source node receives read availability (*white _square*) from all its outgoing channels. Once the message has been sent, the source node deletes all read availability, that is, the white_square.

```
%the outgoing channels from the node
outgoing_channels(["192.168.1.11"]).

% the IP address of the node
ip("192.168.1.13").

% write management by the agent
reaction(
```

```
    in(write(M,_)),
    (operation, invocation),
    (
        outgoing_channels(OC),
        send_mark(OC),
        out(p_write(M))
    )
).

% increase in the number of availabilty arrived
reaction(
    out(pair(white_square, _)),
    completion,
    (
        outgoing_channels(OC),
        in(counter_mark(N)),
        X is N+1,
        length(OC, L),
        K is X mod L,
        out(counter_mark(K))
    )
).

% management of blue_circle sending when the node has received all
    white squares minus those of the SyncDrain nodes connected to
    it
reaction(
    out(pair(white_square, _)),
    completion,
    (
        outgoing_channels(OC),
        length(OC, L),
        rd(counter_mark(M)),
        rd(counter_syncdrain_channels(T)),
        (L-T) =:= M,
        send_blue_circle(OC)
    )
).

% management of token submission when the node has all the white
    squares
reaction(
    out(pair(white_square, _)),
    completion,
    (
        outgoing_channels(OC),
        check(OC),
        rd(p_write(M)),
        send_token(OC,M),
        remove_white_square(OC),
        out(write(M,_))
    )
).

% deletion of the p_write, once the write is completed
reaction(
    in(write(M,_)),
```

```
    completion,
    in(p_write(_))
).

% verification of receipt of a white square (read-availability)
    from all the outgoing channels from the node
check([H]) :- rd(pair(white_square, H)),!.
check([H|T]) :- rd(pair(white_square, H)), check(T).

% sending the token to all outgoing channels
send_token([H],M) :- ip(IP), default @ H : 20504 ?
    out(write(M,IP)), !.
send_token([H|T],M) :- ip(IP), default @ H : 20504 ?
    out(write(M,IP)), send_token(T,M).

% sending the white circle (write availability) to all the
    outgoing channels
send_mark([H]) :- ip(IP), default @ H : 20504 ?
    out(pair(white_circle, IP)), !.
send_mark([H|T]) :- ip(IP), default @ H : 20504 ?
    out(pair(white_circle, IP)), send_mark(T).

% sending of the blue_circle, used to manage the SynckDrain
    channels, to all the outgoing channels
send_blue_circle([H]) :- ip(IP), default @ H : 20504 ?
    out(pair(blue_circle, IP)), !.
send_blue_circle([H|T]) :- ip(IP), default @ H : 20504 ?
    out(pair(blue_circle, IP)), send_blue_circle(T).

% removal of white_square from all outgoing channels
remove_white_square([H]) :- in(pair(white_square, H)), !.
remove_white_square([H|T]) :- in(pair(white_square, H)),
    remove_white_square(T).
```

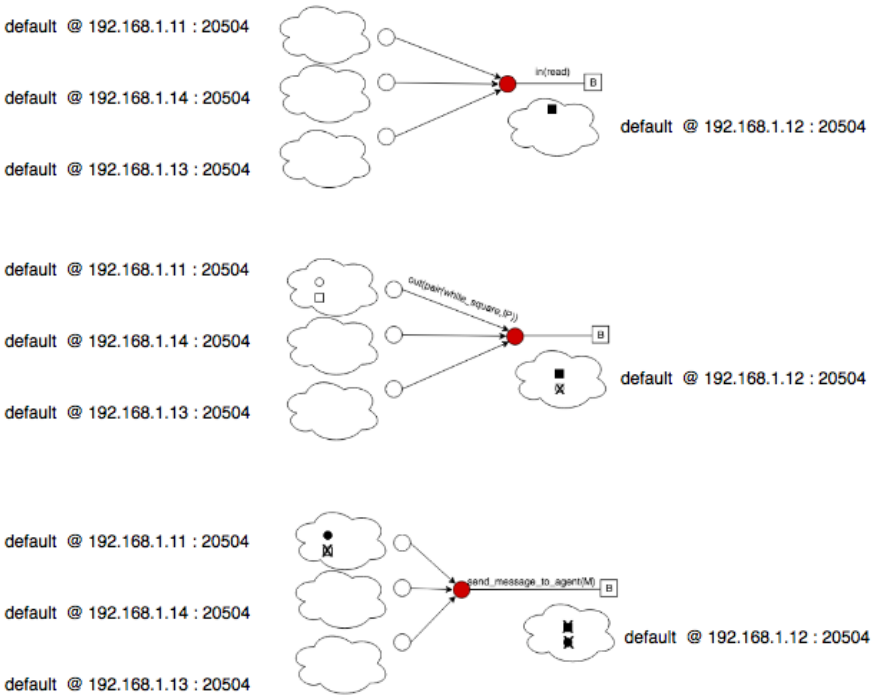
Listing 3: Code needed to configure a tuple center as a source node

3.1.2 Sink Node

Similarly, the behaviour of the Sinc node is characterized by a sequence of operations that can be summarized in 2 steps:

- **send read availability**: when the agent connected to node sends a read operation (*read*), the node notifies its availability to read from the incoming channel that first sent it availability to write (white_circle). The notification to this channel is done by sending a mark (white_square);
- **message reception**: when the Sync node receives the actual message (*write (M, IP)*) it sends it to the agent, it deletes the availability to write (white_circle) and its availability to read in all incoming channels (*ingoing_channels*). If another channel had in the meantime sent a writing request, this request would be managed only at the next reading operation made by the agent.

Figure 2: Sink node semantics



```
%the ingoing channels from the node
ingoing_channels(["192.168.1.11"]).

% the IP address of the node
ip("192.168.1.12").

% read management by an agent when read operation occurs BEFORE a
    write notification
reaction(
    in(read),
    (operation, invocation),
    (
        no(pair(white_circle,_)),
        out(p_read)
    )
).

% read management by an agent when read operation occurs AFTER a
    write notification
reaction(
    in(read),
    (operation, invocation),
    (
        in(pair(white_circle, NTC)),
        ip(IP),
        default @ NTC : 20504  ? out(pair(white_square,IP))
    )
).

% management of the arrival of a writing after a read request
reaction(
    out(pair(white_circle,NTC)),
    completion,
    (
        in(p_read),
        ip(IP),
        default @ NTC : 20504  ? out(pair(white_square,IP)),
        in(pair(white_circle,NTC))
    )
).

% message sending to the agent and  read availability elimination
reaction(
    out(write(M,WIP)),
    completion,
    (
        send_message_to_agent(M),
        ip(IP),
        default @ WIP : 20504 ? in(pair(white_square,IP)),
        out(read)
    )
).

% message elimination from the node, once the read has been
    completed
reaction(
```

```
    in(read),
    completion,
    (
        in(write(_,_))
    )
).

send_message_to_agent(M) :- out(M).
```

Listing 4: Code needed to configure a tuple center as a sink node

3.1.3 Mixed Node

Finally, the behaviour of the Mixed node is characterised by a sequence of operations that can be summarised in 3 steps:

- **reception of the availability to write**: first the node waits for one of its incoming channels to send its availability to write. When this happens, it marks the channel as "first" and then propagates this willingness to write to all its outgoing channels; 
- **reception of availability to read**: after that, the node waits for all its outgoing channels to send their availability to read and, once received all the availability, sends its availability to read to the incoming channel which had been marked as "first";
- **sending the message**: finally, the node waits for the "first" channel to send it the actual message. When this happens, the Mixed node forwards the message to all the outgoing channels, deletes the message and, if there is another write availability by another channel, it marks the channel as "first", otherwise the node goes into waiting.

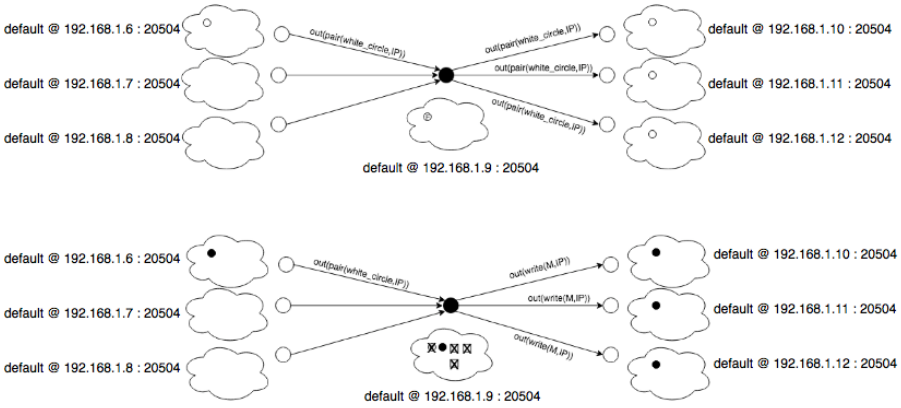
```
%the ingoing channels from the node
ingoing_channels(["192.168.1.7","192.168.1.8"]).

%the outgoing channels from the node
outgoing_channels(["192.168.1.10","192.168.1.11"]).

% the IP address of the node
ip("192.168.1.9").

% management of write availability
reaction(
    out(pair(white_circle, CIP)),
    completion,
    (
        no(first(_)),
        out(first(CIP)),
        outgoing_channels(OC),
        send_mark(OC)
    )
```

Figure 3: Mixed node semantics




```
).


% increase in the number of white squares
reaction(
    out(pair(white_square, _)),
    completion,
    (
        outgoing_channels(OC),
        in(counter_mark(N)),
        X is N+1,
        length(OC, L),
        K is X mod L,
        out(counter_mark(K))
    )
).

% blue circle notification in case the node is connected to one or
    more SyncDrain channels
reaction(
    out(pair(white_square, NTC)),
    completion,
    (
        outgoing_channels(OC),
        length(OC, N),
        counter_mark(M),
        counter_syncdrain_channels(T),
        (N-T) =:= M,
        ip(IP),
        send_blue_circle(OC)
    )
).

% notification of read availability
reaction(
    out(pair(white_square, NTC)),
```

```
    completion,
    (
        outgoing_channels(OC),
        check(OC),
        rd(first(CIP)),
        ip(MIP),
        default @ CIP : 20504  ? out(pair(white_square,MIP))
    )
).

% management of token sending in case there is only ONE write
    availability
reaction(
    out(write(M,CIP)),
    completion,
    (
        outgoing_channels(OC),
        send_token(OC,M),
        in(write(M,_)),
        remove_my_white_squares(OC),
        in(first(CIP)),
        ip(MIP),
        default @ CIP : 20504  ? in(pair(white_square,MIP)),
        in(pair(white_circle,CIP)),
        no(pair(white_circle, CIP2))
    )
).

% management of token sending in case there is MORE writing
    availability
reaction(
    out(write(M,CIP)),
    completion,
    (
        outgoing_channels(OC),
        send_token(OC,M),
        in(write(M,_)),
        remove_my_white_squares(OC),
        in(first(CIP)),
        ip(MIP),
        default @ CIP : 20504  ? in(pair(white_square,MIP)),
        in(pair(white_circle,CIP)),
        rd(pair(white_circle, CIP2)),
        out(first(CIP2)),
        send_mark(OC)
    )
).

% verification of receipt of a white square (read availability)
    from all the outgoing channels from the node
check([H]) :- rd(pair(white_square, H)),!.
check([H|T]) :- rd(pair(white_square, H)), check(T).


% sending the white circle (write availability) to all the
    outgoing channels
send_mark([H]) :- ip(MIP), default @ H : 20504  ?
```

```
    out(pair(white_circle, MIP)), !.
send_mark([H|T]) :- ip(MIP), default @ H : 20504 ?
    out(pair(white_circle, MIP)), send_mark(T).

% sending the token to all outgoing channels
send_token([H],M) :- ip(MIP), default @ H : 20504 ?
    out(write(M,MIP)), !.
send_token([H|T],M) :- ip(MIP), default @ H : 20504 ?
    out(write(M,MIP)), send_token(T,M).

% removal from the node of all white squares received
remove_my_white_squares([H]) :- in(pair(white_square,H)), !.
remove_my_white_squares([H|T]) :- in(pair(white_square,H)),
    remove_my_white_squares(T).

% sending of the blue_circle, used to manage the SyncDrain
    channels, to all the outgoing channels
send_blue_circle([H]) :- ip(IP), default @ H : 20504 ?
    out(pair(blue_circle, IP)), !.
send_blue_circle([H|T]) :- ip(IP), default @ H : 20504 ?
    out(pair(blue_circle, IP)), send_blue_circle(T).
```

Listing 5: Code needed to configure a tuple center as a mixed node

3.2 Basic Channels

There are 6 basic channels that, joined together, can form connectors with more complex behaviours. Below, it will be presented the descriptions of these channels, followed by their respective implementations.

3.2.1 Synch Channel

This channel, compared to the others, is the simplest and most intuitive as regards its operating mechanism. Its goal is to ensure that the Source node, attached to it, is able to send its messages to the Sink node (at the other end of the channel). In order to do this, both the Source and the Sink must be ready, respectively to write and read: as soon as this occurs, the Source node will send the message to the channel that will forward it to the Sink node.

In particular, the behaviour of this channel is defined by the reaction to the following events:

- *out(pair(white_ circle, CIP))*: makes it possible to propagate to the Sink node the possibility of writing a message from the Source node. Propagation occurs by sending a mark (white_circle);
- *out(pair(white_ square, SIIP))*: makes it possible to propagate to the Source node the possibility of reading a message from the Sink node. Propagation occurs by sending a mark (white_square);
- *out(write(M, IP))*: makes it possible to propagate the message to the Sink node. Propagation occurs by sending a write(M,IP).

```
% IP of the Source node attached to the channel
src("192.168.1.12").

% IP of the Sink node attached to the channel
sink("192.168.1.8").

% the IP address of the channel
ip("192.168.1.13").

% management of write availability
reaction(
    out(pair(white_circle,"192.168.1.12")),
    completion,
    (
        sink(IP),
        ip(MIP),
        default @ IP : 20504 ? out(pair(white_circle,MIP))
    )
).

% management of read availability
reaction(
    out(pair(white_square,"192.168.1.8")),
    completion,
    (
        src(IP),
        ip(MIP),
        default @ IP : 20504 ? out(pair(white_square,MIP))
    )
).

% management of message sending
reaction(
    out(write(M,"192.168.1.12")),
    completion,
    (
        in(pair(white_circle,_)),
        sink(IP),
        ip(MIP),
        default @ IP : 20504 ? out(write(M,MIP)),
        in(write(M,_))
    )
).
```

Listing 6: Code needed to configure a tuple center as a Sych channel

3.2.2 LossySync Channel

This channel behaves similar to the Sync channel, but unlike the latter, it allows the Source node to write even when the Sink node is not ready to read, that is when the Reader agent has not yet sent the white square. In this case, the channel loses the message (token) that was to be delivered to the Sink node. Instead, in the event that the Sink node is ready to read, then the forwarding

of the message will be successful.

In particular, the behaviour of this channel is defined by the following events:

- *out(write(M, SIP))*: when the channel receives the message, if there is no read availability of the Sink node, the message is lost, otherwise the message is forwarded to the Sink node;
- *out(pair(white_ circle, SIP))*: when the channel receives a write availability of the Source node, it propagates this notification to the Sink node (out (pair (white_circle, SIIP))) and sends the white square to the Source node to notify the read availability of the channel (out (pair (white_square, IP))).

```
% IP of the Source node attached to the channel
src("192.168.1.13").

% IP of the Sink node attached to the channel
sink("192.168.1.12").

% the IP address of the channel
ip("192.168.1.11").

% management of message sending when there is NO read availability
    of the sink node
reaction(
    out(write(M, "192.168.1.13")),
    completion,
    (
        no(pair(white_square,_)),
        in(write(M,_)),
        in(pair(white_circle,_))
    )
).

% management of message sending when there is read availability of
    the sink node

reaction(
    out(write(M, "192.168.1.13")),
    completion,
    (
        rd(pair(white_square,SIIP)),
        ip(MIP),
        default @ SIIP : 20504 ? out(write(M, MIP)),
        in(write(M,_)),
        in(pair(white_circle,_))
    )
).

% management of sending of write availability notification
reaction(
        out(pair(white_circle, "192.168.1.13")),
    completion,
    (
```

```
        src(SIP),
        sink(SIIP),
        ip(IP),
        default @ SIIP : 20504 ? out(pair(white_circle,IP)),
        default @ SIP : 20504 ? out(pair(white_square,IP))
    )
).

% elimination of read availability
reaction(
    out(write(M,_)),
    completion,
    (
        in(pair(white_square,_))
    )
).
```

Listing 7: Code needed to configure a tuple center as a LossySych channel

3.2.3 FIFO 1 Channel

This channel behaves similar to the LossySync channel but, more than the latter, it allows to memorize the message; in this way, instead of going lost, the message is saved and delivered to the Sink node when this node is ready to receive it. It's called FIFO 1 because it can store at most one message at a time. In particular, the behaviour of this channel is defined by the following events:

- *out(pair(white_ square, SIIP))*: when the channel receives read availability (white_square), it checks whether or not there is a message in its buffer, if present, the message is sent by channel to the Sink node, otherwise the channel keeps the read availability of the Sink node;
- *out(write(M,SIP))*: when the channel receives a message it checks whether or not there is read availability of the Sink node (white_square), if present, the channel sends the message to the Sink node, deletes it and sends its read availability to the Source node (since the buffer has not been filled). In the opposite case it leaves the write availability of the Source node saved;
- *out(pair(white_ circle, SIIP))*: when the channel receives a write availability, it forwards this availability to the Sink node and sends read availability to the Source node.

```
% IP of the Source node attached to the channel
src("192.168.1.13").

% IP of the Sink node attached to the channel
sink("192.168.1.12").

% the IP address of the channel
```

```
ip("192.168.1.11").

%management of availability to read
reaction(
    out(pair(white_square, SIIP)),
    completion,
    (
        in(write(M,_)),
        ip(MIP),
        default @ SIIP : 20504 ? out(write(M,MIP))
    )
).

%management of message sending to the sink node
reaction(
    out(write(M,_)),
    completion,
    (
        sink(SIIP),
        ip(MIP),
        rd(pair(white_square, SIIP)),
        in(pair(white_circle,_)),
        default @ SIIP : 20504 ? out(write(M,MIP)),
        in(write(M,_))
        )
).

%management of availability to write
reaction(
        out(pair(white_circle, "192.168.1.13")),
    completion,
    (
        src(SIP),
        sink(SIIP),
        ip(IP),
        default @ SIIP : 20504 ? out(pair(white_circle,IP)),
        default @ SIP : 20504 ? out(pair(white_square,IP))
    )
).
```

Listing 8: Code needed to configure a tuple center as a Fifo1 channel

3.2.4 Filter(P) Channel

This channel behaves similar to the Sync channel but, unlike the latter, it allows the Sink node to receive only messages that match a certain pattern, messages that do not match are discarded by the channel. In particular, the behaviour of the Filter(P) is defined by the following events:

- *out(pair(white_ circle,SIP))*: when the channel receives a write availability, it forwards this availability to the Sink node;
- *out(pair(white_ square,SIIP))*: when the channel receives a read availability, it forwards this availability to the Source node;

- *out(write(M,SIP))*: when the channel receives the message, check that the message matches one of its patterns. If it matches, the channel forwards the message to the Sink node, otherwise it deletes this message. In any case, the channel eliminates the message, the write availability (white_circle) and the read one (white_square) present in it.

```
% IP of the Source node attached to the channel
src("192.168.1.13").

% IP of the Sink node attached to the channel
sink("192.168.1.12").

% the IP address of the channel
ip("192.168.1.11").

%the list of patterns with which the message must match
pattern(["star","triangle","rhombus"]).

%management of availability to write
reaction(
    out(pair(white_circle,"192.168.1.13")),
    completion,
    (
        sink(SIIP),
        ip(MIP),
        default @ SIIP : 20504 ? out(pair(white_circle,MIP))
    )
).

%management of availability to read
reaction(
    out(pair(white_square,"192.168.1.12")),
    completion,
    (
        src(SIP),
        ip(MIP),
        default @ SIP : 20504 ? out(pair(white_square,MIP))
    )
).

%gestione dell'invio del messaggio quando matcha con uno dei
    pattern

% management of message sending when the message matches with one
    of the patterns, included in the list
reaction(
    out(write(M,"192.168.1.13")),
    invocation,
    (
        pattern(P),
        member(M,P),
        sink(SIIP),
        ip(MIP),
        default @ SIIP : 20504 ? out(write(M,MIP))
    )
```

```
).

% management of message elimination and write availability
reaction(
    out(write(M,_)),
    completion,
    (
        in(write(M,_)),
        in(pair(white_circle,_))
    )
).

% management of the elimination of read availability, if present
reaction(
    out(write(M,_)),
    completion,
    (
        in(pair(white_square,_))
    )
).
```

Listing 9: Code needed to configure a tuple center as a Filter(P) channel

3.2.5 SyncDrain Channel

This channel must ensure that the two Source nodes, attached to it, execute the write_token simultaneously. In order to do so, they must both declare to the channel that they are ready to write (blue_circle). Once the channel receives both the blue circles, it sends the write permission to both nodes (white_square). When the channel receives a message from a node, it deletes this message. In particular, the behaviour of this channel is defined by the following events:

- *out(pair(blue_circle, _))*: when a blue circle arrives, the channel checks that there is also the other one, if so, the channel sends to both its read availability (white_square), otherwise the write availability is saved;
- *out(write(M, IP))*: when the channel receives the message from one of the two Source nodes, it deletes this message and also the blue circle associated to the same Source node.

```
% ingoing nodes in the channel
ingoing_channels(["192.168.1.12", "192.168.1.13"]).

% the IP address of the channel
ip("192.168.1.11").

% management of write availability
reaction(
    out(pair(blue_circle, _)),
    completion,
```

```
    (
        ingoing_channels(IC),
        check(IC),
        send_white_squares(IC)
    )
).

% management of the arrival of a message
reaction(
    out(write(M, IP)),
    completion,
    (
        in(pair(white_circle,IP)),
        in(pair(blue_circle, IP)),
        in(write(M, IP))
    )
).

% verification of the reception of both the blue_circle (write
    availability) of the Source nodes
check([H]) :- rd(pair(blue_circle, H)),!.
check([H|T]) :- rd(pair(blue_circle, H)), check(T).


% sending the white_square to both Source nodes
send_white_squares([H]) :- ip(IP), default @ H : 20504  ?
    out(pair(white_square, IP)), !.
send_white_squares([H|T]) :- ip(IP), default @ H : 20504 ?
    out(pair(white_square, IP)), send_white_squares(T).
```

Listing 10: Code needed to configure a tuple center as a SychDrain channel

3.2.6 AsyncDrain Channel

This channel has a behaviour opposite to the one of the SyncDrain, in fact, its goal is to be sure that the two Source nodes, attached to it, never execute a write operation (*write_token*) simultaneously. In order to do this, the channel must only allow one node at a time to send a message. In particular, the behaviour of the AsyncDrain is defined by the following events:

- *out(pair(white_circle,IP))*: when the channel receives a write availability (white_circle), it verifies that the other Source node has not yet sent one. If the latter availability is not present, the channel sends to the Source node (which has sent its availability) a confirmation message to write (white_square), otherwise only the write availability is saved;
- *out(write(M,IP))*: when the channel receives the message from the Source node (to which the channel itself had given a confirmation message), the channel deletes both the read availability of that node (Source) and the message itself. Then the channel also checks if there is the write availability of the other Source node (which could be waiting), if there is, the channel sends a confirmation message to write (white_square).

```
% ingoing nodes in the channel
ingoing_channels(["192.168.1.12", "192.168.1.13"]).

% the IP address of the channel
ip("192.168.1.11").

%management of availability to write
reaction(
    out(pair(white_circle,IP)),
    completion,
    (
        ingoing_channels(IC),
        not(check(IC)),
        ip(MIP),
        default @ IP : 20504 ? out(pair(white_square,MIP))
    )
).

% management of the message arrival
reaction(
    out(write(M,IP)),
    completion,
    (
        in(pair(white_circle,IP)),
        in(write(M,IP))
    )
).

% management of the presence of another writing availability
reaction(
    out(write(M,IP)),
    completion,
    (
        rd(pair(white_circle,IP2)),
        ip(MIP),
        default @ IP2 : 20504 ? out(pair(white_square,MIP))
    )
).


% verification of receipt of a white_circle (writing availability)
    from both nodes Source
check([H]) :- rd(pair(white_circle, H)),!.
check([H|T]) :- rd(pair(white_circle, H)), check(T).
```

Listing 11: Code needed to configure a tuple center as an AsynchDrain channel

3.3 Initialization

Now, it will be described how to initialise the different nodes and channels before they can be used.

3.3.1 Source Node

- as regards the implementation, enter the correct IPs in "ip (_)" and "outgoing_channels ([_])";
- from CLI (Command Line Interpreter), do out(counter_mark (0)) and out(counter_syncdrain_channels(n)) where n is the number of SyncDrain channels attached to that node (i.e. choose n = 1 if there is only one SyncDrain channel attached to the node);
- optionally, again from CLI, if the node is attached to a LossySync or to a FIFO 1 channel, also do out(pair(white_square, ip)) where ip is the IP of the channel.

3.3.2 Sink Node

As regards the implementation, enter the correct IPs in "ip (_)" and in "ingoing_channels([_])".

3.3.3 Mixed Node

- enter the correct IPs in "ip (_)", "ingoing_channels([_])" and "outgoing_channels([_])";
- do out(counter_mark(0));
- do out(counter_syncdrain_channels(n)) where n is the number of SyncDrain channels attached to that node (i.e. as described above, it is necessary to set n $\bar{1}$ if there is only one SyncDrain channel attached to the node);
- moreover, if the node is attached to a LossySync channel or to a FIFO 1 channel, also do out(pair(white_square, ip)) where ip is the IP of the channel.

3.3.4 Sync Channel

As regards the implementation, enter the correct IPs:

- in ip(ip), src(ip) where ip is the IP of the Source node and in sink(ip) where ip is the IP of the Sink node;
- in the reaction out (pair(white_circle, ip)) where ip is the IP of the Source node, in out (pair (white_square, ip)) where ip is the IP of the Sink node and finally in out(write(M, ip)) where ip is the IP of the Source node.

3.3.5 LossySync e FIFO 1 Channels

As regards the implementation, enter the correct IPs:

- ip(ip);
- src(ip) where ip is the IP of the Source node;
- sink(ip) where ip is the IP of the Sink node.

3.3.6 Filter(P) Channel

As regards the implementation, enter the correct IPs:

- in ip(ip), in src(ip) where ip is the IP of the Source node and in synk(ip) where ip is the IP of the Synk node;
- in the reaction out(pair(white_circle, ip)) where ip is the IP of the Source node, in out(pair(white_square, ip)) where ip is the IP of the Synk node and, finally, in out (write (M, ip)) where ip is the IP of the Source node.

3.3.7 SyncDrain Channel

As regards the implementation, enter the correct IPs in:

- ip(ip);
- ingoing_channels([_]).

3.3.8 AsyncDrain Channel

As regards the implementation, enter the correct IPs in:

- ip(ip);
- ingoing_channels([_]).

Conclusion and Future Work

Both TuCSoN/ReSpecT and REO allow to effectively implement middleware as they provide abstractions by which to define the coordination behaviour between the different components of a distributed system.

Analysing the two models, it is understood that, in TuCSon/ReSpecT it is possible to define the behaviour of a tuple center using the reactions. Reaction operating mechanisms establishes that: when a certain external event (i.e. caused by an Agent) or an interval one (i.e. caused within a the tuple center) occurs, the tuple center can be modified by reacting to the event just happened, according to the tuple center behaviour.

Instead, in REO, differently from what happens in TuCSon/ReSpecT, the coordination policies of the various components are described in terms of channels, nodes and connectors, the latter representing more complex coordination models. Completely abstracting away from the distributed components of the system, the connectors are mapped into REO as *boundary nodes* that interact with the others following the rules of operation of the channels by which they are connected. In REO, the connectors are created starting from the 6 basic channels: Sync, LossySync, FIFO 1, SyncDrain, AsyncDrain and Filter(P).

Basically, the project involved two phases, the former, during which some sets of reactions have been defined to represent well-known examples of connectors (that is Alternator, Sequencer and an Alternator using a Sequencer), the latter, during which the reactions were used to reproduce the behaviour of each base channel so as to obtain the same coordination behaviour between two agents of TuCSon/ReSpecT. These agents, once they have defined the interaction interface between the tuple center and themself, will not have to worry about the coordination policies but only to determine what their goals are.

Since in REO any connector can be defined starting from the base channels, by implementing the latter with the aid of the TuSCoN/ReSPect formalisms, it has been demonstrated, in a practical way, that the expressiveness of the first paradigm is (at least fully understood) equivalent to that of the second, that is: any coordination model expressed in REO can be expressed through TuSCoN/ReSpecT.

This can be already considered a good result on which to base and carry out future projects concerning the study of the expressiveness of these two models. In particular, it would be interesting to create an algorithm which takes as input a REO circuit and provides as output a series of reactions to set up a tuple center: the aim would be to replicate REO coordination policies in TuCSoN/ReSpecT. Furthermore, the study of a complementary project could consist of analysing whether it is possible to reproduce the TuCSoN/ReSpecT model using REO.

References

[1] Omicini A. and Denti E. From tuple spaces to tuple centres. *Elsevier*, 2001.

[2] Omicini A. and Zambonelli F. *Coordination for Internet application development*. 1999.

[3] S.Mariani A.Omicini. Programming interaction with respect. University Lecture, 2017.

[4] S.Mariani A.Omicini. Tuple-based coordination with tucson. University Lecture, 2017.

[5] Farhad Arbab. Reo: A channel-based coordination model for component composition. 2003.

[6] Marco Krauweel. Concurrent and asynchronous javascript programming using reo, 2017.

[7] George A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 11 1995.