ReSpecT\textsuperscript{X}: Programming Interaction Made Easy

Giovanni Ciatto\textsuperscript{1}, Stefano Mariani\textsuperscript{2}, and Andrea Omicini\textsuperscript{1}

\textsuperscript{1} Department of Computer Science and Engineering (DISI)  
ALMA MATER STUDIORUM—Università di Bologna, Italy  
giovanni.ciatto@unibo.it andrea.omicini@unibo.it

\textsuperscript{2} Department of Sciences and Methods for Engineering (DISMI)  
Università degli Studi di Modena e Reggio Emilia, Italy  
stefano.mariani@unimore.it

Abstract. In this paper we present the ReSpecT\textsuperscript{X} language, toolchain, and standard library as a first step of a path aimed at closing the gap between coordination languages – mostly a prerogative of the academic realm until now – and their industrial counterparts. Since the limited adoption of coordination languages within the industrial realm is also due to the lack of suitable toolchains and libraries of reusable mechanisms, ReSpecT\textsuperscript{X} equips a core coordination language (ReSpecT) with tools and features commonly found in mainstream programming languages. In particular, ReSpecT\textsuperscript{X} makes it possible to provide a reference library of reusable and composable interaction patterns.

Keywords: coordination, multi-agent systems, Eclipse IDE, TuCSoN, ReSpecT\textsuperscript{X}

1. Introduction

Many efforts are being devoted in both the industry and the academia to deal with the issue of enabling and governing the interaction space—that is, the dimension of computation defining the admissible interactions among the components of a (possibly concurrent and distributed) system. In the literature it is well understood how such a dimension is conceptually orthogonal to the algorithmic one, thus requiring ad-hoc models and languages \[49\]. While in the industry they often take the form of communication protocols tailored to the particular business domain — e.g., MQTT vs. CoAP for the IoT landscape, FIPA\textsuperscript{3} protocols for multi-agent systems (MAS), REST vs. SOAP for micro-services — in the academia they constitute the subject of the research area known as coordination models and languages \[39\], studying the set of abstractions and mechanisms enabling the management of dependencies amongst computational activities \[32\].

In spite of the number of coordination languages available to date, they are mostly either core calculus, proof-of-concept frameworks, or domain-specific languages for rapid prototyping or simulation, rather than full-fledged programming languages \[17\]. Even though a number of important expressiveness results have been provided \[28\][12][22\] — proving that, in principle, any interaction pattern can be suitably modelled —, their full potential in the engineering of complex distributed systems cannot be really assessed without the corresponding engineering tools. In particular, we believe that the full expressiveness of coordination models and languages should be also measured against the availability of

\textsuperscript{3}http://www.fipa.org/
mature-enough standard libraries and infrastructures actually enabling engineers to build real-world complex systems without, e.g., re-implementing the same interaction patterns over and over again. Also, no suitable toolchain for supporting the increasingly complex task of programming the interaction space is usually provided to developers, resulting in the lack of features typical of state-of-the-art programming languages—like static-checking and live debugging.

On the other hand, in the MAS area – where coordination models and languages are known to be essential to deal with complex interaction patterns \cite{15}, agent-oriented programming (AOP) frameworks are nowadays mostly integrated with mainstream programming languages, and come equipped with all sorts of development tools \cite{45}. As a remarkable example in the field, JADE \cite{5} is a Java-based AOP language and infrastructure equipped with a GUI for remote monitoring of the agents’ lifecycle, an Introspector agent (with a GUI) to debug agents’ inner working cycle, and a Sniffer agent (again, with a GUI) to observe agents’ messaging protocols.

Along this line, the first aim of this paper is to draw the attention of the academic community on the poor support coordination languages provide to the engineering of concurrent and distributed systems such as MAS—when the coordination technology is available at all, of course. So, along with expressive coordination models and languages backed by a sound semantics, the engineering of complex distributed systems also calls for tools and libraries of coordination patterns in order to face complex and mutable interaction requirements.

As a first step to face these issues, we here present the ReSpecT\textsuperscript{X} language, toolchain, and standard library for programming the coordination of MASs as well as distributed applications in general, providing (i) a well founded and expressive semantics, (ii) a number of features supporting the development process, (iii) a library of general purpose, reusable, and composable interaction mechanisms. ReSpecT\textsuperscript{X} builds upon the ReSpecT language \cite{36} inheriting and extending its semantics, while pushing it beyond the limits of other coordination languages through features such as modularity, composability, and tools—with an Eclipse IDE plugin\cite{4} for static-checking, auto-completion, and code generation. Finally, the ReSpecT\textsuperscript{X} Standard Library is conceived as a constantly evolving collection of interaction mechanisms – a few examples of which are shown in the following sections – allowing developers to focus on which particular interaction pattern they need for their agents, instead of how it is actually built.

Accordingly, the remainder of the paper is organised as follows. Section 2 provides the background context that motivates our work, briefly describes a few notable coordination models and languages, and informally describes how the TuCSoN model and technology and the ReSpecT language work. Section 3 presents ReSpecT\textsuperscript{X} and its syntax, and discusses its modularity and composability features. Section 4 showcases some example modules taken from the ReSpecT\textsuperscript{X} Standard Library—purposely selected to highlight their composability. Finally, Section 5 provides some conclusive remarks along with an outlook on possible further developments.

Notice. This paper is an extended version of \cite{16}, firstly presented at the 11th International Symposium on Intelligent Distributed Computing (IDC 2017). Sections 2.1 and 2.2 have been extended to better describe TuCSoN and ReSpecT. Sections 3.2 and 3.4 have
been extended with examples to highlight ReSpecT\textsuperscript{X} improvements over ReSpecT. ReSpecT\textsuperscript{X} Standard Library has been extended with novel reusable mechanisms, described in Section 4.

2. Developing MAS: Computation vs. Interaction

Two prominent examples of agent development frameworks born in the academic world and proficiently transferred to the industry are JADE \cite{5} and Jason \cite{10}: the former is an object-based framework (JADE agents are Java objects) for developing agent-oriented distributed applications in compliance with FIPA standard specifications; the latter is a Java-based implementation of an extension of the AgentSpeak(L) language \cite{43} as well as a BDI agent runtime. Other notable mentions among the many are JADEX \cite{42}, JACK \cite{50}, and SARL \cite{44}, which are industry-ready platforms for developing and running MAS featuring BDI agents. Among the application context where the aforementioned platforms have been actually deployed there are autonomous guidance of unmanned vehicles \cite{48}, smart homes security \cite{11}, surveillance \cite{13}, healthcare \cite{46}, and simulation \cite{27}.

Conversely, examples of coordination languages and infrastructures proficiently exploited in the industrial world are more difficult to find, despite the abundance of well-known and expressive models \cite{17}. To the best of our knowledge, the few coordination technologies that show some degree of maturity – w.r.t. either supporting developers or enabling deployment in real-world systems – are the following:

Reo \cite{3} is a channel-based coordination model that defines how designers can build complex coordinators, called connectors, out of simpler ones. The Reo technology is implemented as a Java library, and comes with a set of related and complementary development tools integrated with the Eclipse IDE, providing for instance a graphical editor and Java code generator \cite{4} plus some data-flow animation and verification tools. Reo has been employed in the field of web services orchestration and composition \cite{30}.

KLAIM/KLAVA \cite{19} is a LINDA-like coordination language for mobile computing focusing on strong code mobility \cite{6}. In KLAIM, both processes and data can be moved across the network among computing environments (tuple spaces), being localities a first-class abstraction meant to explicitly manage mobility and distribution-related aspects. KLAIM is distributed with its own Java code generator \cite{7}, producing sources leveraging on the KLAVA library. No tool is provided to developers, except for the code generator and the KLAVA library itself, and no integration is available with AOP frameworks. KLAIM has been extended by X-KLAIM \cite{8}, but no real-world deployment exists to the best of our knowledge.

LIME \cite{41} is another implementation of LINDA \cite{28} aimed at dealing with both physical (mobile hosts) and logical mobility (code migration) so as to support location-aware computation. LIME focuses in particular on making the federation of tuple spaces transparent, a feature that has been appreciated in the area of wireless sensor networks \cite{18}. LIME is distributed as a Java library providing adaptation layers to different mobile code frameworks and tuple space implementations. Even though the LIME middleware is a mature and robust middleware, it is distributed without tools easing the development and engineering process, and, again, no integration is provided with
AOP frameworks. The only actual deployment of the LIME middleware we found is based on the TeenyLime variant, used to monitor heritage buildings [14].

JavaSpaces/Jini [26] is Oracle’s implementation of LINDA aimed at coordinating distributed Java programs. The focus here is on using Java objects as tuples, storing and retrieving them along with their state, which can be changed by interacting Java programs. Indeed, JavaSpaces provides LINDA-like primitives enabling the insertion (retrieval) of Java objects in (from) object spaces. As objects correspond to LINDA tuples, object spaces correspond to LINDA tuple spaces. The Jini technology is still alive as part of the Apache River project[^5], consisting of an actively maintained and industry-ready middleware implementation. As far as we know, no other support tool is provided, and no integration with AOP frameworks is available.

TuCSoN [40] enriches LINDA tuple spaces with programmability [21]: a TuCSoN tuple centre is a programmable tuple space, and ReSpecT is the language used to program tuple centres [37]. Targeting MAS community, TuCSoN is integrated with both the JADE and Jason AOP frameworks [35]. TuCSoN is distributed as a Java middleware and is actively maintained[^1]—and exploited, for instance, in the healthcare field [23]. Before ReSpecT[^X] and its ecosystem, it provided minimal development supporting facilities.

In the remainder of this paper we focus on TuCSoN, and in particular on the ReSpecT language used to program tuple centres, since ReSpecT[^X] is built on top of it.

2.1. Structuring the interaction space with TuCSoN

TuCSoN [40] is a tuple-based coordination model available as a Java-based middleware [1], providing coordination as a service [47] to the agents in a MAS—or, more generally, processes in a distributed system. Following (and extending) the archetypal LINDA semantics, TuCSoN agents coordinate by means of coordination primitives allowing them to read (rd), insert (out), consume (in), or test for absence of (no) first-order logic tuples within LINDA-like tuple spaces—see Fig. 1. Agents actions are synchronised thanks to LINDA suspensive semantics [28] affecting the so-called getter primitives—rd, in, and no. This means, for instance, that an agent trying to consume (in) a tuple matching a given template from a tuple space succeeds only after such a tuple has been found in the tuple space—typically inserted by an out. TuCSoN extends LINDA with the predicative, bulk, and probabilistic version of the aforementioned primitives. Predicative primitives (rdp, inp, and nop) are not susceptible to suspensive semantics, therefore represent predicates about the state of the tuple spaces. Bulk primitives are used by agents willing to insert (out_all), read (rd_all), or consume (in_all) multiple tuples all at once within tuple spaces. Finally, probabilistic primitives (urd, uin, and uno) let agents read / consume tuples probabilistically, refining LINDA non-determinism with a uniform probability distribution [33]. Since TuCSoN makes no assumption on the agents inner architecture and capabilities, any Java program can be enabled to exploit its coordination services via TuCSoN API. As a result, TuCSoN works as a general-purpose coordination medium for distributed systems in general.

TuCSoN tuple spaces are actually tuple centres [37] because they are enhanced with a behaviour specification—that is, a program specifying how the tuple space itself must

[^5]: http://river.apache.org
TuCSoN nodes are spread on a network of hosts, tuple centres are hosted by these nodes, and agents interact by means of TuCSoN primitives.

TuCSoN expects tuple centres behaviour specifications to be expressed in the ReSpecT language, shortly described in next subsection.

TuCSoN is fully integrated with JADE and Jason by properly harmonising LINDA suspensive semantics and TuCSoN invocation modes with JADE and Jason concurrency models, and comes equipped with a few tools for monitoring, debugging, manual testing, and inspection of the interaction space. Thus, TuCSoN represents a seldom case of mature-enough coordination infrastructure actually viable as a solid option for coordinating real-world industrial applications – for instance, to replace message-based with stigmergic coordination in those scenarios where loose coupling of interacting entities is required (e.g., in smart homes and eHealth scenarios), with a further benefit for those already exploiting JADE or Jason.

2.2. Programming the interaction space with ReSpecT

ReSpecT is a Prolog-based declarative language for defining tuple centre behaviour specification. Each specification is composed by one or multiple specification tuples aimed at intercepting the events involving the local tuple centre and to provide some ad-hoc action to be performed as reaction. Specification tuples are a special kind of first-order logic tuples whose form is

\[ \text{reaction}((\text{Event}), (\text{Guards}), (\text{Body})) \]

where:
Fig. 2. Pictorial representation of the triggering mechanism of ReSpecT reactions: a coordination-related event (i.e., invocation of a TuCSoN primitive) triggers one or more reactions, whose body executes provided that the guards hold true. When a reaction is executing, it may trigger other reactions, in a sort of chain.

⟨Event⟩ is the triggering event of the reaction—that is, the coordination-related event whose occurrence triggers evaluation of the reaction, like the invocation of a coordination primitive on the local tuple centre, or the local time reaching a given instant

⟨Guards⟩ is the (conjunction of) guard predicate(s) about the properties of the triggering event that must hold true for the reaction to be actually executed—thus enabling fine-grained control over reactions selection

⟨Body⟩ is the reaction body—that is, the sequence of Prolog computations and ReSpecT primitives representing the intended actions to be performed as a response to each occurrence of the triggering event

Fig. 2 graphically represents the triggering mechanism of ReSpecT reactions.

As an example, let us suppose ACME Inc. is leveraging TuCSoN/ReSpecT for its workflow automation framework. In its simplest variant, the framework expects tasks to be scheduled on the todolist@acme.com tuple centre in the form of tuples matching the template task(ARGV). A number of worker-agents are responsible for the execution of scheduled tasks. An agent may take charge of a task by consuming the corresponding task(ARGV) tuple, e.g., by invoking in(task(ARGV)) on the todolist tuple centre. ACME Inc. does not allow its worker-agents to be idle, so it programs the tuple centre to always provide a default task to idle workers by means of the following reaction, lazily producing an unbounded amount of default tasks.

```
reaction{
    in(task(_)), % reaction triggered whenever a task will be consumed
    { invocation, % reaction executed before any task is (possibly) consumed
      from_agent % the invocation must come from an agent
    },
    { no(task(_)), % ensure no other task is available
      event_source(WorkerName), % get the name of the worker
      default_action_for_agent(WorkerName, Action), % Prolog computation
      out(task(Action))) % produce the default task
},
```
Such a reaction is triggered whenever an agent invokes the \texttt{in(task(\_))} primitive (the invocation is the triggering event), and both the \texttt{invocation} and \texttt{from_agent} guards evaluate to \texttt{true} given the current state of the tuple centre. The \texttt{invocation} guard holds true until the operation is actually served – thus before any tuple is actually removed from the tuple centre –, while the \texttt{from_agent} guard holds true only if the invocation comes from an agent—i.e. not from a tuple centre. Once triggered, the reaction should abort if some tuple in the form \texttt{task(\_)} occurs in the tuple centre because it would represent an higher-priority task ready to be consumed by the requesting agent. In other words, the reaction should continue only if no tuple having the aforementioned form exists within the tuple centre. This is indeed the aim of the \texttt{no(task(\_))} line, which would simply fail if any of such tuples exist. Analogously to Prolog semantics, the failure of a predicate within some reaction body causes the failure of that reaction as a whole: \textsc{ReSpecT} failed reactions leave the tuple centre state unchanged, as better explained below.

The rest of the reaction body simply retrieves the name of the entity provoking the triggering event, an information which is always available within reaction bodies by means of the \texttt{event_source/1} predicate, computes the default \texttt{Action} for agent \texttt{WorkerName} (by means of \texttt{ACME Inc.'s default_action_for_agent/1} predicate) and schedules a task containing that action—by means of \texttt{out(task(Action))}.

\textbf{Reactions chaining.} If some tuple centre's behaviour specification contains more than one reaction, an event may trigger more than one reaction. Moreover, coordination primitives invocations could occur within specification tuples, too, thus further reactions may be recursively triggered. So, what happens when multiple reactions are triggered (either directly or recursively) by an event? According to \textsc{ReSpecT} semantics \cite{35}, reactions are executed sequentially in a non-deterministic order, atomically, and with a transactional semantics. In short, this implies that (i) reactions – triggered within the same tuple centre – are executed one at a time with no overlapping whatsoever (sequentially), (ii) each reaction either succeeds or fails as a whole (atomically), (iii) a failed reaction causes no effect at all (they are transactions)—i.e., each side-effect provoked (resp. reaction triggered) by a failing reaction is reverted (resp. cancelled).

As an example, imagine ACME Inc. is extending its workflow framework with the capabilities of (i) automatically scheduling a task after another has been accomplished, (ii) or, automatically requiring a pre-condition task to be accomplished before another is actually scheduled.

To this end, it endows the \texttt{todolist} tuple centre with the following reactions.

```prolog
reaction(
  out(end(Action)),
  completion,
  (rd(doAfter(Action, Next)),
   out(task(Next))
  )
).
```

As in the previous example, the application assumes agents to take charge of a task by con-
suming the corresponding `task(Action)` tuple, and to declare their accomplishment of that task by inserting a `end(Action)` tuple into the same tuple centre. The leftmost reaction intercepts the insertion of tuples matching `end(Action)`, i.e., the accomplishment of a task. If a tuple `doAfter(Action, Next)` exists within the tuple centre, the reaction produces the corresponding `task(Next)`, otherwise it simply fails leaving the tuple centre unchanged. Analogously, the rightmost reaction is triggered by the insertion of a newly scheduled `task(Action)`. If a tuple `doBefore(Action, Prev)` exists within the tuple centre, the reaction schedules `task(Prev)` – which should be accomplished first – and states that `task(Action)` should be rescheduled only after `task(Prev)` has been accomplished. If no such a tuple exists, the reaction fails leaving the tuple centre unchanged. The reaction would fail also if a `doAfter(Prev, Action)` exists within the same tuple centre. This would leave `task(Action)` as the just-scheduled task.

Notice that the rightmost reaction may be triggered by a successful execution of both the leftmost one and itself—because both reaction bodies have a line matching the rightmost reaction triggering event `out(task(Action))`. This would be the case, for instance, of a Git-based workflow scenario where tuples `doBefore(commit, run_tests)` and `doBefore(push, commit)` both occur within the `todolist` tuple centre and someone tries to schedule the `push` task. The rightmost reaction would be triggered, thus causing the `commit` task to be scheduled instead of `push`, and the tuple `doAfter(commit, push)` to be produced. Again, the rightmost reaction would be triggered by the scheduling of task `commit`, causing the `run_tests` task to be scheduled instead, and the `doAfter(run_tests, commit)` tuple to be produced. Then, once the tests have been successfully executed and the `end(run_tests)` tuple published, the leftmost reaction would be triggered, removing the `end` tuple and producing the `task(commit)` one. Of course, this would trigger the rightmost reaction again, which is undesirable since it would prevent the `commit` task to be scheduled. But this time, the `no(doAfter(Prev, Action))` would make the rightmost reaction fail, reverting all side effects it caused—such as that stemming from line `in(task(Action))`.

The mechanism exemplified above is called “reaction chaining”, and enables the implementation of expressive and flexible event-driven coordination in TuCSoN [34].

Meta-coordination primitives. Agents interacting through TuCSoN tuple centres can dynamically manipulate the set of reactions characterising a tuple centre behaviour by means of `meta-coordination` primitives. Such primitives – with both their suspensive and predicative variants – are aimed at `dynamically` inserting (`out_s`), consuming (`in_s` and `inp_s`), reading (`rd_s` and `rdp_s`) or checking for absence of (`no_s` and `nop_s`) specification tuples within the tuple centre they are invoked upon. Similarly to specification tuples, meta-coordination primitives are ternary predicates in the form

\[ \langle \text{MetaPrimitive} \rangle (\text{Event}, (\text{Guards}), (\text{Body})) \]

which can be invoked either by the agents or by tuple centres themselves, within other reactions bodies.

In particular, meta-coordination primitives can be used within ReSpecT programs to postpone the “activation” of some reaction by lazily inserting the corresponding specification tuple with the `out_s` primitive, or “deactivating” some other reaction by removing
the corresponding specification tuple with the \texttt{in\_s} primitive. Of course, other usage scenarios are supported, too, such as checking whether a reaction for a given event exists (by means of \texttt{rd\_s}) or not (by means of \texttt{no\_s}), or reacting to the insertion/removal of some specification tuple, thus providing great malleability of both the tuple centre and its behaviour.

**ReSpecT shortcomings.** The ReSpecT Virtual Machine (VM henceforth) is the Prolog-based engine responsible for on-the-fly interpretation (triggering, evaluation, and execution) of specification tuples, which may be either statically programmed by human developers at design-time, or injected into a running TuCSoN system through meta-coordination primitives, either by coordinating agents or tuple centres themselves. Statically programmed specifications must be grouped into a single, monolithic behaviour specification file which can be loaded onto a tuple centre, removing any previously existing specification tuple from that tuple centre—thus replacing its behaviour. Conversely, dynamic injection of specification tuples into a tuple centre may occur at any time by means of the aforementioned meta-coordination primitives.

Despite ReSpecT being a Turing-powerful language [22] capable of capturing most of other coordination models and actively exploited in a number of academic and industrial projects [20,23], the lack of features typical of mainstream programming languages – e.g., modularity, composability, concise syntax, debugging support etc. –, the lack of a suitable toolchain assisting developers through the code-debug-fix loop, as well as the lack of a library providing reusable and composable implementations of well-established coordination mechanisms – such as publish-subscriber services – hinders its diffusion and adoption in industrial environments. Indeed, until now, even if every interaction pattern could be virtually implemented by properly programming TuCSoN tuple centres with ReSpecT, developers should re-implement basic mechanisms from scratch over and over again.

Accordingly, in the following section we present the ReSpecTX language, toolchain, and standard library. ReSpecTX is an extension of ReSpecT dealing with the aforementioned issues by re-designing the language to support both modularity and a more concise syntax, and providing suitable IDE tools aimed at intercepting programming issues as soon as possible (e.g., by means of static-checking and IDE integration).

**Related work.** ReSpecT – thus ReSpecTX – shares features with other approaches exploiting some form of Event-Condition-Action (ECA) rules.

For instance, within the scope of Event-Based Systems [25], ECA rules represent a well established pattern for expressing the business logic of a component. ECA rules are essentially triplets having the form “When \texttt{(Event\_Type)} occurs, If \texttt{(Condition)} holds, Perform \texttt{(Action)}”. It is evident that ReSpecT reactions do adhere to this pattern: event types correspond to TuCSoN primitives, conditions to guards, and actions to arbitrary ReSpecT / Prolog computations.

ReSpecT reactions also presents analogies with Complex-Event-Processing (CEP) systems [31], where the ability to capture events and generate new ones in response are of primary importance. What ReSpecT lacks w.r.t. CEP systems is a first-class support for event-related operators such as \texttt{after}, \texttt{before}, \texttt{coincides}, \texttt{during}, and so on, which are formally defined in [2], and generally employed within CEP systems to recognise complex
events. Nevertheless, it is easy to build such operators using ReSpecT itself—as shown by the above examples in ReSpecT.

Finally, it may be argued that ReSpecT reactions are really close to AgentSpeak(L) plans, whose purpose is, i.e., to handle believes addition/removal from BDI agents’ belief base. Nevertheless, their target is very different: whereas the former are meant to program a tuple centre, that is, the coordination artefacts governing the interactions between agents, the latter are intended to program BDI agents—that is, the proactive entities interacting with each other [38].

3. ReSpecTX: eXtended ReSpecT

ReSpecTX empowers ReSpecT with a few crucial features, enhancing the language itself and adding the necessary tooling, thoroughly described in the upcoming subsections.

**Modularity** — Unlike ReSpecT monolithic files, ReSpecTX program definitions can be split into different modules to be imported in a root specification file, enabling and promoting code decomposition and reuse as well as development of code libraries.

**Development tools** — ReSpecTX programs are written through an editor distributed as an Eclipse IDE plugin, featuring syntax highlighting, static error checking, code completion, and code generation of ReSpecT specification files and Prolog theories.

**Syntactic sugar** — ReSpecTX adds special guard predicates testing presence/absence of tuples without side effects (e.g., actual consumption of tuples), and adopts a more concise syntax, stressing the procedural semantics of reaction bodies, for the benefit of developers not familiar with declarative languages such as Prolog.

**Standard library** — ReSpecTX comes with a standard library of modules implementing general purpose coordination mechanisms, utilities, and interaction patterns easing, for instance, the creation of networks of tuple centres, the spreading of tuples over such networks, the exploitation of a tuple centre as a publish-subscribe service, the scheduling of delayed or periodic activities, etc.

It is worth to highlight how ReSpecT remains the underlying language actually exploited for coordination by the TuCSoN middleware. Indeed, ReSpecTX comes with a code generator – automatically invoked by the Eclipse IDE – aimed at producing a low-level monolithic ReSpecT specification file. This way, the TuCSoN infrastructure has not been modified at all, and ReSpecTX is totally interoperable with legacy ReSpecT specifications. Though still in beta stage and not yet available as a stand-alone Eclipse distribution, the ReSpecTX development project is already publicly available as open source code [6] – installation instructions are also provided –, allowing a test IDE to be experimented. Fig. 3 shows a screenshot of the Eclipse environment. Notice that both ReSpecTX and ReSpecT languages are supported.

3.1. Syntax overview

A ReSpecTX script consists of a single file containing a ⟨Module⟩ definition. Modules are of two sorts: library modules, conceived to be reused by other modules, and specifications. Both contain the definition of the ⟨Reaction⟩s implementing the coordination

---

6 [http://bitbucket.org/gciatto/respectx](http://bitbucket.org/gciatto/respectx)
mechanisms and policies to achieve a given goal in a specific application domain. In both cases, the script must provide a qualified name enabling both its identification and localization within the file system, analogously to Java packages and class names.

Each module may declare an arbitrary number of reactions as well as Prolog clauses \(\langle\text{PrologExpression}\rangle\). This implies that a module may contain arbitrary logic theories which can be exploited within reaction bodies to implement interaction rules. Finally, in the same way as in ReSpecT, ReSpecTX reactions have a triggering \(\langle\text{Event}\rangle\) (e.g., the invocation of some primitive or the local time reaching a given instant), an optional \(\langle\text{Guards}\rangle\) expressing conditions about the current state of the tuple centre, or the ReSpecT VM, or the event itself, and a \(\langle\text{Body}\rangle\) composed by ReSpecT primitives invocations or Prolog predicates / functors calls.

Table II below shows a detailed description of ReSpecTX grammar. The most interesting features are thoroughly discussed in the following subsections.

### 3.2. Modularity, re-usability, composability

The ReSpecT VM expects reactions to be loaded on a tuple centre as a single monolithic script. Consequently, the development of non-trivial coordination logic in ReSpecT is often uncomfortable and error-prone due to the size of the specification script. Moreover, developers wanting to reuse their tested and correctly working reaction set can only rely on copy & paste. ReSpecT in fact provides no linguistic abstraction to partition specifications. Even if further reactions can be dynamically added to (or removed from) a tuple centre by means of meta-coordination primitives, reusability is nonetheless hindered.

ReSpecTX overcomes such a limitation by providing two explicit scoping mechanisms at the language level: modules and specifications. ReSpecTX library modules --
Table 1. ReSpecTX language grammar

```
(Module) ::= module (QualifiedName) { (ModuleBody) }
  | specification (QualifiedName) { (ModuleBody) }
(ModuleBody) ::= include (QualifiedName)
  | { PrologExpression }.
  | (Reaction)
  | (ModuleBody) (ModuleBody)

(Reaction) ::= (OptVirtual) reaction (OptName) to (Body)
(OptName) ::= ε | @ ⟨ReactionName⟩ ⟨PrologVarList⟩
(OptVirtual) ::= ε | virtual

(InlineReaction) ::= reaction (Body)
(Body) ::= (Event) { (PrologCode) }
  | (Event) : (Guards) { (PrologCode) }

(Event) ::= (PrimitiveExecution)
  | time (Instant)
(PrimitiveExecution) ::= ⟨Primitive⟩ ⟨TupleTemplate⟩ ⟨OptReturning⟩
(OptReturning) ::= ε | returning ⟨ListTemplate⟩

(Primitive) ::= out | rd | rdp | in | inp | no | nop
  | urd | urdp | uin | uinp | uno | unp
  | out_all | in_all | rd_all | no_all | out_s
  | rd_s | rdp_s | in_s | inp_s | no_s | nop_s

(Guards) ::= (Guard) | (Guard), (Guards)
(Guard) ::= ? ⟨TupleTemplate⟩ | ! ⟨TupleTemplate⟩
  | invocation | completion | success
  | failure | endo | exo

(PrologCode) ::= ⟨PrimitiveInvocation⟩
  | ⟨ObservationAtom⟩
  | ⟨PrologExpression⟩
(ObservationAtom) ::= ⟨ObservationView⟩ ⟨ObservationInfo⟩
(ObservationView) ::= current | event | start
(ObservationInfo) ::= predicate | tuple | source | target | time
(PrimitiveInvocation) ::= ⟨Primitive⟩ ⟨TupleTemplate⟩ ⟨OptReturns⟩
(OptReturns) ::= ε | returns ⟨ListTemplate⟩
```
declared by means of the `module` keyword – are meant to wrap logically-related reactions and Prolog predicates within the same file, together providing general-purpose and reusable behaviours. Of course, each module may rely on pre-existing ones to provide its functionalities. For instance, in the following sections we show how a module implementing tuple dissemination over a network of tuple centres can be built on top of the module providing interconnection facilities.

More precisely, a module definition contains an arbitrary number of (i) statements of the form `include ⟨QualifiedName⟩`, recursively importing all the reactions defined in the referenced module into the current one; (ii) Prolog facts and rules providing the computations needed to realize potentially articulated behaviours; (iii) reactions, realising the coordination policies provided by the module. `ReSpecT X` applications are wrapped within a single file – declared with the `specification` keyword – that the `ReSpecT X` compiler parses and translates into the aforementioned monolithic file expected by the `ReSpecT VM`, by composing all the `ReSpecT X` reactions defined therein and in each included module. The above mechanisms straightforwardly support modularity, by enabling the creation of libraries of modules implementing general coordination mechanisms and policies suitable to be used, and composed together, in different contexts.

To support those scenarios where an additional behaviour must be injected into some tuple centre as a response to an external event, reactions can be dynamically inserted by means of meta-coordination primitives, as already mentioned. This soon leads to a problem analogous to the callback-hell making the specification code difficult to read, understand, and debug. To prevent such issues, reactions in `ReSpecT X` can be declared as `virtual`—which implies they must provide a name. Virtual reactions are explicitly meant to be referenced as the `actual` argument of meta-coordination primitives. They provide no behaviour until they are “activated” by an `outs`. The combination of `virtual` and referenceability helps avoiding spaghetti-code in reactions, thus improving their readability. Besides names, virtual reactions may specify unbound variables as `formal` arguments, making their behaviour parametrizable.

As an example, suppose a malicious developer wants to hack ACME Inc.’s workflow system enabling employees to `silently` delegate their personal tasks to their colleagues, possibly promising something in return. The hacker guesses that `EmployeeName`’s personal tasks are in the form `personal_task(Action, EmployeeName)` and that their execution is acknowledged by tuples in the form `end(Action, EmployeeName)`. Then, he/she simply injects the following reactions into the `todolist` tuple centre (the `ReSpecT` version is shown on the left, the `ReSpecT X` version on the right):

---

7 [http://callbackhell.com/](http://callbackhell.com/)
Such a reaction (on the left) enables an employee $E$ to delegate his/her tasks to someone else ($D$), by `out`ing a tuple in the form `delegate(E, D)`. To do so, the reaction dynamically specifies two more reactions by means of the `out_s` meta-coordination primitive: the first one states what should happen whenever a `personal_task(A, E)` is scheduled targeting $E$, while the second one states what should happen once the delegate $D$ acknowledges the execution of the task with `end(A, D)`. In the former case, the task is re-scheduled in order to target $D$, and the original schedule is removed. In the latter case, the acknowledgement of $D$ is replaced with another one, `end(A, E)`, targeting $E$.

The ReSpecT\textsuperscript{X} variant of this reaction (on the right) is more compact and concise, leveraging on two virtual and named reactions whose only purpose is to be referenced as the argument of the aforementioned `out_s` meta-coordination primitive.

### 3.3. Toolchain: static-checking, code completion, code generation

ReSpecT lacks development tools: thus, for instance, ReSpecT programmers become aware of syntactic or semantic errors only at run-time, by receiving a failure response when trying to load an incorrect specification file. More subtly, syntactically correct but inconsistent specifications – containing, e.g., a reaction having contradictory guards – would be silently accepted. ReSpecT\textsuperscript{X} overcomes the issue by empowering ReSpecT with Eclipse IDE integration (in the form of a plugin) featuring static-checking, code completion, and generation.

The Eclipse IDE plugin is implemented by exploiting the Xtext framework\[^8\] which provides a few handy features common in mainstream programming languages, such as syntax coloring, code completion, static-checking while writing code, and automatic generation of ReSpecT code—there included Prolog predicates and functors. Syntax coloring and code completion straightforwardly move ReSpecT\textsuperscript{X} closer to mainstream programming languages. The static-checker needs some deeper discussion, stemming from the peculiarities of tuple-based coordination languages and of declarative languages too. For instance, in declarative untyped languages such as Prolog there is no declaration phase regarding variables: they are simply used when required. This complicates spotting common problems such as useless variables. Further complicating things, for tuple-based co-

\[^8\] <http://eclipse.org/Xtext/>
ordination languages understanding what is admissible, useful, or even meaningful at the language level heavily depends on the state of the tuple centre when the reaction specification is executed: for instance, attempting to consume a tuple when no matching one is available cannot be checked statically. For the above reasons, the amount of checks that can be done are limited w.r.t. traditional programming languages.

ReSpecT IDE now detects: (i) repeated reactions within the same specification, i.e., reactions triggered by the same triggering event and enabled by the same guards; (ii) inconsistent temporal constraints; (iii) bad-written URLs or TCP port numbers (e.g., reserved ones); (iv) singleton variables within a reaction, that is variables appearing only once, which may hide a typo; (v) contradictory ReSpecT guards preventing reaction execution regardless of the context, as defined in table below:

<table>
<thead>
<tr>
<th>Reference Guard</th>
<th>Contradictory Guard</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>invocation</td>
<td>completion</td>
<td>—</td>
</tr>
<tr>
<td>endo</td>
<td>exo</td>
<td>—</td>
</tr>
<tr>
<td>intra</td>
<td>inter</td>
<td>—</td>
</tr>
<tr>
<td>success</td>
<td>failure</td>
<td>—</td>
</tr>
<tr>
<td>from_agent</td>
<td>from_tc</td>
<td>—</td>
</tr>
<tr>
<td>to_agent</td>
<td>to_tc</td>
<td>—</td>
</tr>
<tr>
<td>before(T1)</td>
<td>after(T2)</td>
<td>T1 &gt;= T2</td>
</tr>
<tr>
<td>?X</td>
<td>!Y</td>
<td>X = Y, ground(X)</td>
</tr>
</tbody>
</table>

Indeed, a reaction declared with both guards invocation and completion is worthless, because it would be enabled only if the ReSpecT VM is simultaneously in two mutually exclusive phases, which is impossible. The same rationale drives the functioning of the static-checker w.r.t. other contradictory guards. The last rows of the table above prevents developers from writing inconsistent temporal constraints and looking for both the presence of a tuple and its absence.

### 3.4. Syntax enhancements

ReSpecT declarative syntax – inherited from the Prolog programming language – may be considered both a blessing and a curse. Despite declarativity being often a desirable feature in programming languages, stressing what computations should do instead of how to do it, Prolog syntax can easily become verbose, especially when the same set of computation must be performed on lists of tuples. Moreover, the procedural interpretation of Prolog code is particularly evident within ReSpecT reaction bodies, where order of side effects performed on the local tuple centre is indeed relevant and not negligible. Accordingly, ReSpecT IDE provides an hybrid style syntax in order to be more handy to write and easier to read:

- primitive invocations are unary prefix operators: `out T` equals `out(T)`
- identifiers of TuCSoN tuple centres have a human-readable syntax
- the `if C then T else F` construct is introduced as a more familiar alternative to Prolog’s `(C -> T ; F)` expressions in order to easily provide branching computations within reaction bodies
- ReSpecT’s observation predicates – such as `event_time/1`, `event_source/1`, etc. – become context-sensitive atoms in ReSpecT IDE. This makes inspecting the reaction context more handy. For instance, the following ReSpecT reaction (on the left)
– used to log the insertion of tuples matching some(Template) – appears far less concise than its ReSpecTX counterpart (on the right)

```prolog
reaction{
  out(some(Template)),
  true, {
    event_time(Time),
    event_predicate(Pred),
    event_tuple(Tuple),
    event_source(Source),
    event_target(Target),
    log(Time, Pred, Tuple, Source, Target)
  }
}
```

– The procedural nature of ReSpecTX reactions is stressed by their syntax. For instance, the following snippet shows the ReSpecTX version of the reaction allowing agents to infinitely consume a default task when they are idle (presented in Subsection 2.2):

```prolog
reaction @idle_task
  to in task(_) : invocation, from_agent, !task(_) {
    default_action_for_agent(event_source, Action),
    out(task(Action))
  }
```

The reaction name is meant to be referenced likewise java method names, and the reaction body is delimited by curly braces to syntactically group the part of the specification subject to procedural interpretation (as for Java methods’ body). In between lies the triggering event along with the (optional) list of guards: the former not only decides when to trigger the reaction, but also enables to bind variables to actual values – through Prolog unification mechanism – thus may map to a Java method invocation (there including parameters), whereas the latter is peculiar of ReSpecT and has no intuitive mapping with mainstream programming languages. Summing up, the reaction name alongside with the triggering event and the guards constitute altogether the “method signature” of a ReSpecTX specification, playing a similar role of Java methods’ signature.

Furthermore, ReSpecTX also provides some syntactic sugar reducing the boilerplate code w.r.t. ReSpecT specifications. For instance, special guards checking the presence (?<TupleTemplate>) or absence (!<TupleTemplate>) of a tuple are provided. In the near future, we plan to extend the set of available special guards with arbitrary predicates about the state of the local tuple centre.

4. ReSpecTX standard library

In the following subsections we focus on ReSpecTX modularity feature to showcase how reusability of reactions is straightforwardly enabled by encapsulation and composition. Accordingly, we introduce some modules from the ReSpecTX Standard Library.\(^9\)

\(^9\) https://bitbucket.org/gciatto/respectx-standard-library
briefly describing their functioning and the use case they target. Then we show how simple behaviours, provided by the aforementioned modules, can be composed into more articulated ones by means of ReSpecTX features.

Other ready-to-use ReSpecTX modules are available in the standard library – e.g., logging predicates supporting reaction debugging, functional-like predicates easing data manipulation, or fork-join facilities for processes coordination – while others are currently under development and testing.

4.1. Building reusable mechanisms

We now describe a number of ReSpecTX modules encapsulating the logic for some basic mechanisms – such as scheduling of periodic activities, handling tuples multiplicity, interconnecting tuple centres to form a network, or exploiting a tuple centre as a publish-subscribe service – provided as ready-to-use coordination laws from the ReSpecTX standard library.

Scheduling periodic activities. Listing 1.1 shows the ReSpecTX code implementing module rsp.timing.Periodic, making it possible to schedule a periodic activity, which is a building block for several distributed design patterns, i.e., decay or resilient spreading [24].

```
module rsp.timing.Periodic {

  reaction to out start_periodic(Period, Activity) : exo, completion {
    in start_periodic(Period, Activity),
    if no periodic_context(_, _, Activity) then {
      out periodic_context(Period, 0, Activity),
      out tick(Activity)
    }
  }  

  reaction to out tick(Activity) : endo, ?periodic_context(Period, _, Activity) {
    in tick(Activity),
    NextTickInstant is Period + current_time,
    out_s @next_tick(NextTickInstant, Activity)
  }  

  virtual reaction @next_tick(T, A) to time(T) : endo, ?periodic_context(Period, _, TickNumber, Activity) {
    in periodic_context(Period, TickNumber, A),
    NextTickNumber is TickNumber + 1,
    out periodic_context(Period, NextTickNumber, A),
    out A,
    out tick(A)
  }  

  reaction to out stop_periodic(Activity) : exo, completion {
    in all stop_periodic(Activity),
    in_all periodic_context(_, _, Activity),
    in_all tick(Activity)
  }  

}
```

Listing 1.1. The Periodic module
Activities are represented by an Activity tuple: the module takes care of emitting the Activity tuple once every Period milliseconds; then, if a reaction has tuple out(Activity) as triggering event, its body would be executed periodically. By producing a tuple of kind start_periodic(Period, Activity) (or, respectively, stop_periodic(Period, Activity)), the periodic activity is started (resp. stopped) causing

- reification of a periodic_context (if none already exists) tracking the period, number of executions carried out, and the Activity tuple—to allow for several periodic activities to be executed concurrently
- emission of the tick(Activity) tuple to trigger scheduling of the next insertion, thus creating the desired loop—through insertion of a new instance of the virtual reaction next_tick

Whenever next_tick is executed: (i) the periodic_context is updated; (ii) the Activity tuple is emitted; (iii) and a tuple tick(Activity) is emitted to (re)trigger the loop.

For instance, if an organisation leveraging on ACME Inc.’s workflow system wants to schedule the ‘cleanup_building’ task once per day, it must simply insert a start_periodic(86400000, task(cleanup_building)) tuple within the todolist tuple centre on system deploy. In this case, the Activity tuple to be generated once per day has the form task(cleanup_building).

Decorating tuples with multiplicity. There are application contexts where it is convenient to decorate tuples with their multiplicity, increasing performance of getter operations; for instance, in the case tuple spaces are used as biochemical solutions simulators [29]. There, tuples are considered as molecules floating in a chemical solution (the tuple centre), tuple templates as chemical species, and multiplicity of tuples their chemical concentration.

Listing 1.2 shows the rsp.biochemical.Concentration module, providing library support to such a form of decorated tuples: (i) the tuple centre is forced to behave like a set instead of a multi-set for tuples matching the conc(Tuple) template, which are stored as conc(Tuple, Concentration); (ii) whenever a tuple conc(Tuple) is emitted (resp. consumed) the corresponding Concentration is increased (resp. decreased).

Essentially, the module makes ordinary TuCSoN primitives (e.g., out, in, rd, no, etc.) conform to their usual contract despite tuples’ decoration:

- if a species conc(Tuple) already exists, a tuple conc(Tuple, Concentration) and exactly one copy of conc(Tuple) are stored until Concentration ≤ 0—so as to make rd, rdp, no, and nop function as usual. An invocation of either inp or in conc(Tuple) would just decrease the Concentration value
- if Concentration ≤ 0 for a given species no conc(Tuple) tuple is stored, to preserve usual functioning of rd, rdp, no, and nop. An invocation of inp conc(Tuple) would fail, whereas invoking in conc(Tuple) would decrease the Concentration of that species while tracking down waiting agents

\[1\text{ day} = 24 \times 60 \times 60 \times 1000\text{ ms} = 86400000\text{ ms}\]
Dynamically creating a network of tuple centres. A notion of neighborhood for a tuple centre can be introduced by making it aware of other tuple centres. To this purpose we propose the \textit{rsp.net.Neighborhood} module, shown in Listing 1.3, which essentially makes the conventions adopted to build a connection between any two tuple centres explicit, and provides a simple protocol enabling an agent to stimulate such a connection.

More precisely, we assume that a tuple centre \texttt{Name1 @ Address1 : Port1} is connected to another tuple centre \texttt{Name2 @ Address2 : Port2} if the latter one contains a tuple \texttt{nbr(Name1, Address1, Port1)}. So, for the connection to be symmetric there must exist a tuple \texttt{nbr(Name2, Address2, Port2)} on the former one, too. A connection from a tuple centre to another may be interrupted simply by removing the reference to the latter from the former, by means of an ordinary \texttt{in}. We also assume that each tuple centre composing the network contains a tuple \texttt{self(MyName, MyAddress, MyPort)}, making the tuple centre aware of its own identity. Under such hypotheses, the \texttt{neighborhood/1} predicate can be employed by reaction bod-

Listing 1.2. The \texttt{Concentration} module
ies to retrieve the current list of neighbours of the tuple centre evaluating it, whereas the out_to_all/2 can be used to insert a given tuple on a number of tuple centres.

```prolog
module rsp.net.Neighborhood {  
neighborhood(Neighbors) :-  
   rd_all nbr(_, _, _) returns Neighbors.  
self(MyName, MyAddress, MyPort) :-  
   rdp self(MyName, MyAddress, MyPort).  
neighbor('@'(TCName, ':'(Address, Port)), nbr(TCName, Address, Port)).  
neighbor('@'(TCName, Address), nbr(TCName, Address, 20504)).  
neighbor(TCName, nbr(TCName, "localhost", 20504)).  
out_to_all(_, []).  
out_to_all(Tuple, [nbr(TCName, Address, Port) | Others]) :-  
   out Tuple on TCName @ Address : Port,  
   out_to_all(Tuple, Others).
}
```

Listing 1.3. The Neighborhood module

A tuple centre Name @ Address : Port may request a connection to another one by sending tuple want_connect(Name @ Address : Port). Any tuple centre receiving a tuple of such a sort, reacts by producing a local nbr(Name, Address, Port) tuple, reads for its self(MyName, MyAddress, MyPort) tuple, and sends back a nbr(MyName, MyAddress, MyPort) to the tuple centre which started the connection protocol.

An agent may make a tuple centre Name1 @ Address1 : Port1 start a connection protocol with another tuple centre Name2 @ Address2 : Port2 by invoking out connect_to(Name2, Address2, Port2) on Name1 @ Address1 : Port1. In this case, the former tuple centre would react by outing a tuple want_connect(Name1, Address1, Port1) on the latter one, thus beginning the interconnection protocol.

**Tuple centres as publish-subscribe intermediaries.** TuCSoN tuple centres can be programmed to work as persistent brokers in the publish-subscribe (pub-sub from now on) interaction pattern. Agents in a pub-sub architecture may play two roles: publishers or subscribers. Publishers are in charge of perceiving event occurrences and publish their notifications to the rest of the system, regardless of other agents actually being interested in such information. Subscribers may be interested in one (ore more) particular class of event notifications, so they just subscribe to that class and then wait for notifications of
interest, regardless of the agents publishing them. If the system provides a persistent brokering service, subscribers need not to be on-line when event notifications are published.

The `rsp.interaction_patterns.PublishSubscribe` module implements a persistent pub-sub mechanism that can be used to make a brokering service out of a TuCSoN tuple centre. The module script is shown in Listing 1.4 and the way it works is described below.

The `PublishSubscribe` module expects publishers to publish facts on a tuple centre by `out`ing a tuple having the form \texttt{publish(Fact)}, where \texttt{Fact} is an event notification payload. Analogously, subscribers can subscribe to a particular event notification \texttt{template Topic} by `out`ing a tuple having the form \texttt{subscribe(Topic)}. Whenever a \texttt{Fact} is published, the tuple centre would store a notification for each subscriber whose \texttt{Topic} is matching that \texttt{Fact}. As soon as a subscriber is ready to handle the next notification, it can retrieve it by invoking `in notify(Topic)`. If a matching fact had already been published, the subscriber consumes it, otherwise its request is suspended until some is published.

Listing 1.4. The `PublishSubscribe` module

```plaintext
module rsp.interaction_patterns.PublishSubscribe {  
    reaction to out subscribe(Topic) : completion, from_agent {  
        in_all subscribe(Topic),  
        out subscription(Topic, event_source)  
    }  
    reaction to out publish(Fact) : completion, from_agent {  
        in_all publish(Fact),  
        rd_all subscription(Fact, _) returns Subscriptions,  
        store_publications(Fact, Subscriptions)  
    }  
    reaction to out publication(F, A) : endo, ?waitfor(F, A) {  
        in waitfor(F, A),  
        in publication(F, A),  
        out notify(F)  
    }  
    reaction to in notify(Fact) : invocation, from_agent {  
        if rd publication(Fact, event_source) then (  
            in publication(Fact, event_source),  
            out notify(Fact)  
        ) else (  
            out waitfor(Fact, event_source)  
        )  
    }  
    store_pub(Fact, subscription(Fact, Recipient)) :-  
        out publication(Fact, Recipient).  
    store_pub(_, subscription(_, _)).  
    store_publications(_, []).  
    store_publications(F, [S | Ss]) :-  
        store_pub(F, S),  
        store_publications(F, Ss).  
}
```
More precisely, the PublishSubscribe module makes the tuple centre intercept productions of tuples matching `subscribe(Topic)`, replacing them with tuples in the form `subscription(Topic, Agent)`, where `Agent` is the subscriber’s name. The tuple centre also intercepts the production of tuples matching `publish(Fact)`, replacing it with as many `publication(Fact, Recipient)` tuples as the amount of tuples matching `subscription(Fact, Recipient)` contained into the tuple centre and as many `notify(Fact)` as the number of subscribers waiting for their `in notify(Fact)` invocation to be handled. In this way, whenever a subscriber invokes `in notify(Fact)`, it will eventually retrieve the notification of published events it is interested in (if any is ever published).

4.2. Articulated behaviours as composition of mechanisms

We now show a few examples of articulated behaviours achieved by means of modules composition, like for instance the “decay” module, consuming tuples periodically to decrease their relevance over time, or the “spreading” module, diffusing tuples over a network of tuple centres. The two newly-created modules, despite producing an articulated behaviour, still appear to have a very concise representation thanks to ReSpecT constructs allowing reuse of pre-existing modules.

Information relevance decaying with time. As a simple yet paradigmatic example of reusability through encapsulation and composition, the `rsp.biochemical.Decay` module shown in Listing 1.5 implements the “decay” mechanism often found in nature-inspired and/or adaptive coordination models [39] whenever the relevance of some information must decrease as time progresses. It relies on the other modules just described: periodically, tuples are consumed regardless of whether they are either individual or decorated ones.

The module works as follows: (i) either an agent or a tuple centre emits the `start_periodic(Period, decay(Template))` tuple to trigger periodic emission of the therein defined tuple `decay(Template)`; (ii) such a tuple represents an activity to be performed once every `Period` milliseconds, namely `decaying` the multiplicity of all tuples matching the provided template; (iii) the tuple centre reacts to the insertion of the `decay(Template)` tuple by reducing the concentration of decorated tuples, if `Template = conc(Tuple)`, or by consuming a tuple matching `Template`, otherwise.

```
module rsp.biochemical.Decay {
  include rsp.biochemical.Concentration
  include rsp.timing.Periodic
  decay_one(conc(Tuple)) :- remove_one_if_any(Tuple).
  decay_one(Something) :- in Something.
  reaction to out decay(Something) {
    inp decay(Something),
    decay_one(Something)
  }
}
```

Listing 1.5. The Decay module
Spreading tuples over a network of tuple centres. The `rsp.net.Spreading` module, shown in Listing 1.6, implements a simple “spreading” mechanism based on the aforementioned interconnection capabilities provided by the `rsp.net.Neighborhood` module.

```plaintext
module rsp.net.Spreading {
  include rsp.net.Neighborhood
  include rsp.biochemical.Concentration
  produce_one(conc(Tuple)) :- put_one(Tuple).
  produce_one(Something) :- out Something.
  reaction to out spreading(Tuple) : exo, invocation {
    if no spreading(Tuple) then {
      produce_one(Tuple),
      neighborhood(Neighbors),
      out_to_all(spreading(Tuple), Neighbors)
    } else {
      in_all spreading(Tuple)
    }
  }
}
```

**Listing 1.6. The Spreading module**

The module makes the tuple centre react to the insertion of a `spreading(Tuple)` tuple, by leveraging the `out_to_all/2` and `neighborhood/1` predicates mentioned above. The reaction is in charge of sending a copy of `spreading(Tuple)` tuple to each tuple centre composing the local neighbourhood, and of creating a local copy of `Tuple`. The reaction has no side effects if a `spreading(Tuple)` tuple already exists within the tuple centre when a newer one is received. So, the occurrence of a tuple of the form `spreading(Tuple)` is the termination condition for the spreading process of `Tuple` over a network of tuple centres with an unknown topology.

Again, we provide an implementation of the “spreading” mechanism transparently supporting either decorated or ordinary tuples. In the decoration case, the spreading affects tuples concentration, increasing it, whereas the effect for ordinary tuples is a greater multiplicity.

The reader may have noticed how such a module actually implements a fragile spreading mechanism. In fact, since the diffusion of information only occurs once – i.e., after a tuple matching `spreading(Tuple)` has been inserted into (any node of) a network of tuple centres –, only those tuple centres which are currently on line eventually receive the information spread. ReSpecTX easily enables developers to implement a “resilient spreading” mechanism – continuously diffusing information on the network – by composing the aforementioned `rsp.net.Spreading` and the `rsp.timing.Periodic` modules. One may imagine, for instance, to replace the reaction in Listing 1.6 with the following lines:

```plaintext
include rsp.timing.Periodic
reaction to out spreading(Tuple) : endo {
  if no spreading(Tuple) then {
    produce_one(Tuple),
  } else {
    in spreading(Tuple)
  }
  neighborhood(Neighbors),
  out_to_all(spreading(Tuple), Neighbors)
}
```
The two reactions above are enough to make the spreading mechanism resilient: whenever an external agent initiates a spreading process by *out*ing a spreading(Tuple), a periodic activity is started causing the Tuple information to be actually spread on the neighbourhood, *once per second*. So, even if some node in the neighbourhood is not currently on-line, it will eventually receive the information spread within a few seconds after its re-connection.

In a similar way, the ReSpecTX standard library supports and encourages composition of the modules mentioned in this section to build increasingly complex coordination pattern, from the more classic message-passing or publish-subscribe ones, to the bio-inspired, pheromone-based, or stigmergic ones—as classified in [24].

## 5. Conclusions and Further Work

In this paper we presented the ReSpecTX language, toolchain, and standard library for programming the interaction space of distributed systems, aimed at closing the gap between the conceptual advancement of coordination languages and their technological maturity, so as to promote their widespread adoption. To this end, ReSpecTX has been equipped with a few crucial features paving the way toward full integration with mainstream programming languages and toolchain—modularity, static error checking, and Eclipse IDE integration being the most notable ones. ReSpecTX also comes with a code generator producing low-level ReSpecT code specifying the behaviour of TuCSoN tuple centres. Thus, ReSpecTX is fully interoperable with the TuCSoN middleware and its legacy applications. In particular, this paper focuses on the ReSpecTX Standard Library, being—to the best of our knowledge—the first attempt of providing a library of reusable coordination mechanisms supporting several general-purpose interaction patterns.

Next steps planned to further improve ReSpecTX and its ecosystem include the development of more sophisticated debugging tools, possibly providing a tighter integration with mainstream IDE environments such as Eclipse or JetBrains’ IntelliJ. Finally, for ReSpecTX to have an impact, the set of ready-to-use composable coordination mechanisms provided by its Standard Library should be constantly extended to cope with an increasing number of application scenarios and their typical interaction patterns.

### References

1. TuCSoN: Home Page (2008), [http://tucson.unibo.it](http://tucson.unibo.it)


Giovanni Ciatto is a PhD student in “Data Science and Computation” at DISI, the Department of Informatics, Science and Engineering of the Alma Mater Studiorum - Università di Bologna, Italy. He holds a master’s degree in Computer Science and Engineering, cum laude, from the same Department. His current research interests include blockchain and Cloud technologies, coordination middleware for multiagent systems, intelligence for the IoT.
Stefano Mariani is assistant professor of Computer Science at the University of Modena and Reggio Emilia. He got his PhD in Computer Science from the University of Bologna in 2016. He has been involved in the EU FP7 Project SAPERE, and currently is involved in EU H2020 Project CONNECARE. His research interests include: coordination models and languages, agent-oriented technologies, pervasive computing, self-organisation mechanisms, socio-technical systems.

Andrea Omicini is Full Professor at DISI, the Department of Computer Science and Engineering of the Alma Mater Studiorum–Università di Bologna, Italy. He holds a PhD in Computer and Electronic Engineering, and his main research interests include coordination models, multi-agent systems, intelligent systems, programming languages, autonomous systems, middleware, simulation, software engineering, pervasive systems, and self-organization. On those subjects, he published over 300 articles, edited a number of international books, guest-edited several special issues of international journals, and held many invited talks and lectures at international conferences and schools.

Received: January 11, 2018; Accepted: September 4, 2018.