Programming the Interaction Space Effectively with ReSpecTX

Giovanni Ciatto* Stefano Mariani† Andrea Omicini*
stefano.mariani@unimore.it†, {andrea.omicini, giovanni.ciatto}@unibo.it*

Department of Science and Methods of Engineering, Università di Modena e Reggio Emilia †
Department of Computer Science and Engineering, Università di Bologna *

School of Electrical Engineering
Belgrade, October 12th 2017

Outline

Context

Both industry and academia developing methods to govern the *interaction space* [Wegner, 1997]

- *communication protocols* in industry
  - MQTT vs. CoAP → IoT landscape
  - FIPA[1] protocols → multi-agent systems (MAS)
  - REST vs. SOAP → micro-services
- *coordination models and languages* in academia [Omicini and Viroli, 2011]
  - control driven → Reo [Arbab, 2004]
  - data driven → LINDA [Gelernter, 1985]
  - hybrid → ReSpecT [Omicini, 2007]

[1] `http://www.fipa.org/`

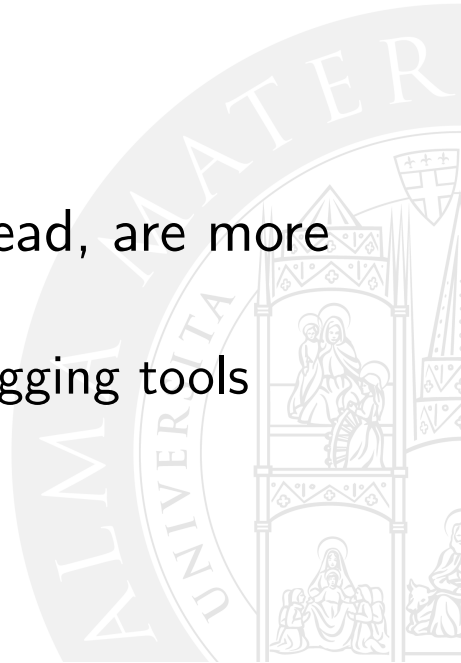
Motivation

- Coordination languages mostly are
  - core calculus
  - proof-of-concept frameworks
  - domain-specific languages for rapid prototyping / simulation

$\Rightarrow$ **no toolchain**, basically

- ✗ *no* Integrated Development Environment (IDE)
- ✗ *no* debugging
- ✗ *no* static-checking
- ✗ *no* code-completion

- *Agent-oriented Programming* (AOP) frameworks, instead, are more mature
  - JADE has many administration, monitoring, and debugging tools [Bellifemine et al., 2007]
  - *Jason* has an IDE and a monitoring / debugging tool [Bordini et al., 2007]



Goal

- *Close the gap* between maturity of AOP languages and coordination frameworks
- Focus on supporting development process

We present the ReSpecT$\mathbb{X}$ **language and toolchain**

✓ ReSpecT$\mathbb{X}$ builds upon ReSpecT [Omicini, 2007]
- \+ *modularity*
- \+ Eclipse *IDE* plugin
  - ✓ *static-checking*
  - ✓ *auto-completion*
  - ✓ *code generation*
- \+ imperative-style syntactic sugar

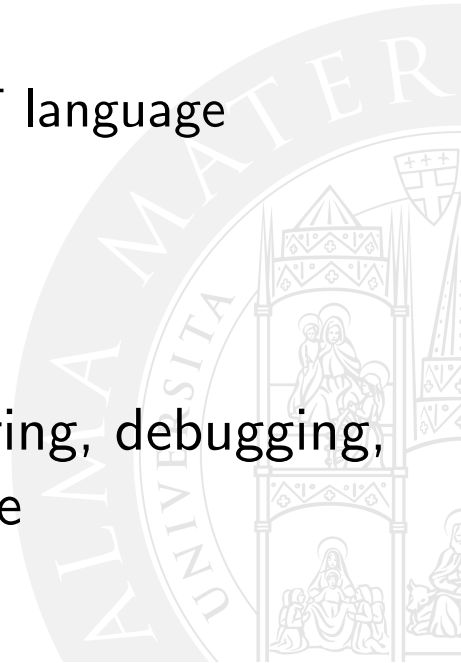
Outline

The TuCSoN Coordination Infrastructure

- TuCSoN [Omicini and Zambonelli, 1999] is a model and infrastructure providing *coordination as a service* [Viroli and Omicini, 2006] to a MAS in the spirit of the archetypal LINDA model
  - Tuples are stored in *tuple centres* [Omicini and Denti, 2001]
    - ⇒ tuple spaces enhanced with a *program* specifying how the tuple space must react to coordination-related events
  - Tuple centres' programs are expressed in the ReSpecT language [Omicini, 2007]

✓ TuCSoN is fully integrated with JADE and *Jason* [Mariani and Omicini, 2016]

✓ TuCSoN comes equipped with *a few* tools for monitoring, debugging, manual testing, and inspection of the interaction space



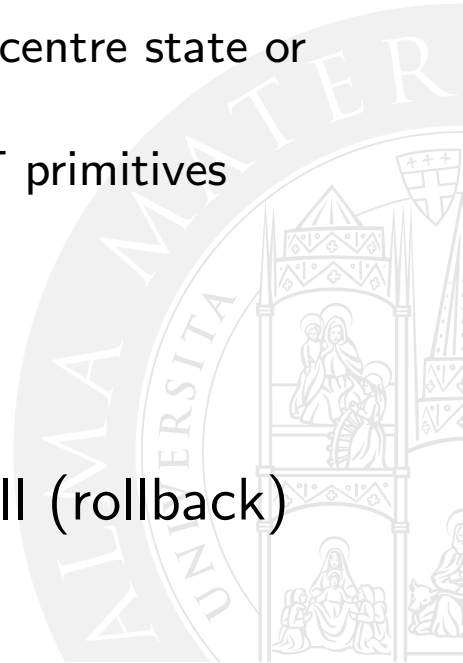
The ReSpecT Coordination Language I

- ReSpecT [Omicini, 2007] is a Prolog-based language for *programming* tuple centres
- A ReSpecT program is a set of *specification tuples* (or, *reactions*)
  - first-order logic tuples of the form `reaction`$(\langle E \rangle, \langle G \rangle, \langle R \rangle)$
    - $\langle E \rangle$ = *triggering event* = coordination primitive
    - $\langle G \rangle$ = (set of) *guard predicate*(s) = conditions on tuple centre state or triggering event
    - $\langle R \rangle$ = *reaction body* = Prolog computations + ReSpecT primitives
- Each reaction is executed

  sequentially one at a time, no overlapping

  atomically either succeed or fail as a whole

  transactionally a failed reaction causes no effects at all (rollback)



The ReSpecT Coordination Language II

Example: infinite tuples

```
reaction( in(inf(T)), invocation,
  ( no(inf(T)),
    out(inf(T)) )
).
```

- `in(inf(T))` = triggering event
- `invocation` = guard (`true` *before* the operation is served)
- `( no(inf(T)), out(inf(T)) )` = reaction body

Outline

Highlights

modularity ReSpecT𝕏 programs can be split in *modules* imported in a root *specification* file

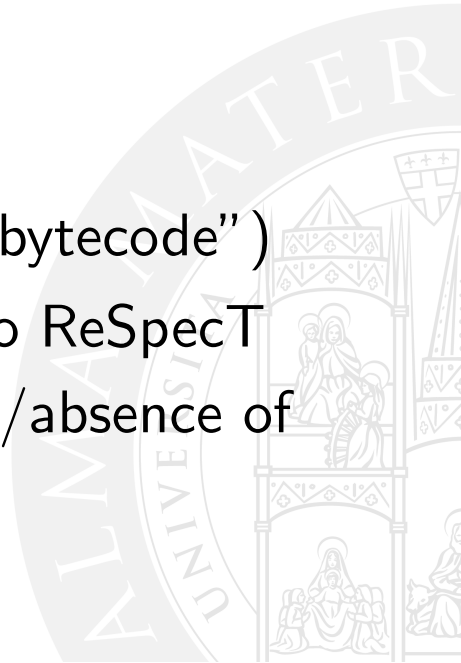
- ⇒ **code reuse**
- ⇒ code libraries

toolchain *Eclipse IDE plugin*

- ✓ syntax highlighting
- ✓ **static error checking**
- ✓ code auto-completion
- ✓ code generation (plain ReSpecT is the "bytecode")

syntax ReSpecT𝕏 adds convenient syntactic sugar to ReSpecT

- special guard predicates testing presence/absence of tuples *without* side effects
- *imperative style* syntax



Outline

Modularity ⇒ Re-usability, and Composability

- A `module` definition contains an arbitrary number of:
  - `include` ⟨*QualifiedName*⟩, which *imports* reactions defined in the referenced module
  - Prolog facts and rules
  - ReSpecT𝕏 reactions
- A `specification` does the same, but is also translated by the ReSpecT𝕏 compiler in a plain ReSpecT program (directly *executable* by TuCSoN)
- Reactions can be decorated with a `@`⟨*ReactionName*⟩ tag
  - ⇒ *referenceable* by *meta-coordination primitives* (primitives with a ReSpecT𝕏 specification tuple as argument)
- Tagged reactions can be further decorated with keyword `virtual`
  - ⇒ *inactive* until the meta-coordination primitive activates them

Example: Scheduling Periodic Activities I

The module outputs the `A` tuple *once every* `P` milliseconds ⇒ the activity represented by a reaction with `out(A)` as triggering event gets executed periodically

- ✓ the `module` can be imported and *reused at will*
- ✓ tagged, virtual reaction exploited

Example: Scheduling Periodic Activities II

```
1  module rsp.timing.Periodic {
2    reaction out startP(P, A) : completion {
3      current_time(Now)
4      out context(P, 0, Now, A),
5      out tick(A)
6    }
7    reaction out tick(Activity) : endo, ?context(P, _, _, A) {
8      current_time(Now),
9      NextTick is Now + P,
10     out_s @next_tick(NextTick, A)
11   }
12   @next_tick(T, A)
13   virtual reaction time(T) : ??context(_, Tick, _, A), ??tick(A) {
14     NextTick is Tick + 1,
15     current_time(Now),
16     out context(P, NextTick, Now, A),
17     out A,
18     out tick(A)
19   }
20 }
```

Example: Change Tuple Centre Nature I

- The module forces the tuple centre to behave like a *set*, instead of a multi-set, for tuples matching the `set(Tuple)` template
- `set(Tuple)` tuples are stored as `set(Tuple, M)` where `M` is their *multiplicity*
- Whenever a tuple `set(Tuple)` is emitted (consumed) the corresponding `M` is automatically increased (decreased)

Example: Change Tuple Centre Nature II

```
1  module rsp.lang.SetBehaviour {
2      put_one(Tuple) :-
3          if nop set(Tuple, _) then out set(Tuple, 1)
4          else if inp set(Tuple, M) then (
5              NextM is M + 1,
6              out set(Tuple, NextM),
7              if (NextM > 1) then inp set(Tuple)
8          ) else fail.
9      reaction out set(Tuple) : completion, exo { put_one(Tuple) }
10     remove_one(Tuple) :-
11         if inp set(Tuple, M) then (
12             if (M > 0) then (
13                 NextM is M - 1,
14                 out set(Tuple, NextM),
15                 in_all set(Tuple) returns _,
16                 if (NextM > 0) then out set(Tuple)
17             )
18         ).
19     reaction inp set(Tuple) : completion, exo { remove_one(Tuple) }
20 }
```

Example: Do Both :) I

- The module implements the "decay" mechanism often found in nature-inspired coordination models [Omicini and Viroli, 2011] by reusing and composing the previous modules
  - ✓ `startP(P, decay(TT))` tuple triggers periodic emission tuple `decay(TT)`
  - ✓ then, reaction `@decay` starts triggering in loop, creating the decay effect

Example: Do Both :) II

```
1  module rsp.lang.Decay {
2      include rsp.lang.Concentration
3      include rsp.timing.Periodic
4
5      decay_one(Something) :-
6          if (Something = set(Tuple)) then (
7              remove_one(Tuple)
8          ) else (
9              inp Something
10         ).
11     @decay
12     reaction out decay(Something) {
13         inp decay(Something),
14         decay_one(Something)
15     }
16
17     % Remember that startP(P, decay(TT)) tuple
18     % triggers periodic emission of decay(TT)
19 }
```

Example: Do Both :) III

ReSpecT𝕏 **standard library**[a] provides other modules to build increasingly complex *coordination patterns*, such as *gossiping* in a mobile network, *stigmergic coordination*, and others [Fernandez-Marquez et al., 2012]

[a] `http://bitbucket.org/gciatto/respectx-standard-library`

Outline

Toolchain: Static-checking, Code Completion, Code Generation I

- Eclipse IDE plugin[2] implemented in the `Xtext` framework[3] (as ReSpecT𝕏 itself)
- Handy features common in mainstream programming languages
  - syntax coloring
  - *code completion*
  - automatic generation of ReSpecT code
  - **static-checking**

Toolchain: Static-checking, Code Completion, Code Generation II

- The static checker detects
  - *duplicate* reactions in a module (recursively), i.e. reactions having same $\langle E \rangle$ and $\langle G \rangle$
  - *inconsistent* temporal constraints
  - bad-written URLs or TCP port numbers (i.e. reserved ones);
  - *singleton variables*, that is, variables appearing only once in a reaction
  - *contradictory* ReSpecT guards

`invocation, completion`	`endo, exo`
`intra, inter`	`success, failure`
`from_agent, from_tc`	`to_agent, to_agent`
`?X, !Y if X = Y, ground(X)`	`before(T1), after(T2) if T1 >= T2`

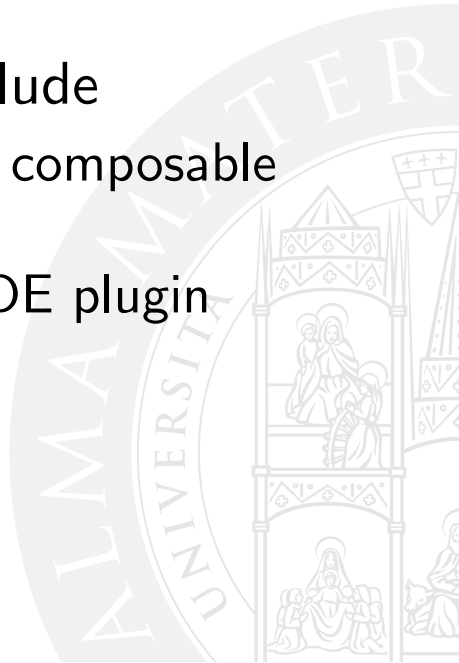
[2] Already publicly available as open source code at `http://bitbucket.org/gciatto/respectx`

[3] `http://eclipse.org/Xtext/`

Outline

Conclusion & Ongoing Work

- ReSpecT$\mathbb{X}$ is a **first step** in *closing the gap* between coordination languages and mainstream programming languages
  - ✓ modularity
  - ✓ static error checking
  - ✓ automatic code generation
- Next steps to further improve ReSpecT$\mathbb{X}$ maturity include
  - development of a *rich standard library* of ready-to-use composable coordination mechanisms
  - distribution of ReSpecT$\mathbb{X}$ as *ready-to-install* Eclipse IDE plugin
  - *improve static checker*



References I


Arbab, F. (2004).
Reo: A channel-based coordination model for component composition.


Bellifemine, F. L., Caire, G., and Greenwood, D. (2007).
Developing Multi-Agent Systems with JADE.
Wiley.


Bordini, R. H., Hübner, J. F., and Wooldridge, M. J. (2007).
Programming Multi-Agent Systems in AgentSpeak using Jason.
John Wiley & Sons, Ltd.


Fernandez-Marquez, J., Marzo Serugendo, G., Montagna, S., Viroli, M., and Arcos, J. (2012).
Description and composition of bio-inspired design patterns: a complete overview.


Gelernter, D. (1985).
Generative communication in Linda.


Mariani, S. and Omicini, A. (2016).
Multi-paradigm coordination for MAS: Integrating heterogeneous coordination approaches in MAS technologies.
CEUR Workshop Proceedings, 1664:91–99.

References II


Omicini, A. (2007).
Formal ReSpecT in the A&A perspective.


Omicini, A. and Denti, E. (2001).
From tuple spaces to tuple centres.


Omicini, A. and Viroli, M. (2011).
Coordination models and languages: From parallel computing to self-organisation.


Omicini, A. and Zambonelli, F. (1999).
Coordination for Internet application development.


Viroli, M. and Omicini, A. (2006).
Coordination as a service.
Fundamenta Informaticae, 73(4):507–534.
Special Issue: Best papers of FOCLASA 2002.


Wegner, P. (1997).
Why interaction is more powerful than algorithms.

Programming the Interaction Space Effectively with ReSpecT$\mathbb{X}$

Giovanni Ciatto* Stefano Mariani† Andrea Omicini*
stefano.mariani@unimore.it†, {andrea.omicini, giovanni.ciatto}@unibo.it*

Department of Science and Methods of Engineering, Università di Modena e Reggio Emilia †
Department of Computer Science and Engineering, Università di Bologna *

School of Electrical Engineering
Belgrade, October 12th 2017