Programming the Interaction SpaceEffectively 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
1 Premises

2 Background

3 ReSpecTX: eXtended ReSpecT
   - Modularity, Composability
   - Toolchain

4 Conclusion & Ongoing Work
Outline

1. Premises

2. Background

3. ReSpecTX: eXtended ReSpecT
   - Modularity, Composability
   - Toolchain

4. Conclusion & Ongoing Work
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

⇒ 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\textsuperscript{X} language and toolchain

- ReSpecT\textsuperscript{X} builds upon ReSpecT [Omicini, 2007]
  - modularity
  - Eclipse IDE plugin
    - static-checking
    - auto-completion
    - code generation
  - imperative-style syntactic sugar
Outline

1 Premises

2 Background

3 ReSpecTX: eXtended ReSpecT
   - Modularity, Composability
   - Toolchain

4 Conclusion & Ongoing Work
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.
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 $\text{reaction} (\langle E \rangle, \langle G \rangle, \langle R \rangle)$
  - $\langle E \rangle = \text{triggering event} = \text{coordination primitive}$
  - $\langle G \rangle = (\text{set of}) \text{guard predicate(s)} = \text{conditions on tuple centre state or triggering event}$
  - $\langle R \rangle = \text{reaction body} = \text{Prolog computations + ReSpecT primitives}$

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

\[
\text{reaction ( } \text{in(}\inf(T)\text{)}, \text{invocation,}
\text{ ( no(}\inf(T)\text{)}, \\
\text{ out(}\inf(T)\text{)) ) .}
\]

- \text{in(}\inf(T)\text{)} = \text{triggering event}
- \text{invocation} = \text{guard (true before the operation is served)}
- \text{( no(}\inf(T)\text{)}, \text{ out(}\inf(T)\text{)) )} = \text{reaction body}
Outline

1. Premises
2. Background
3. ReSpecTX: eXtended ReSpecT
   - Modularity, Composability
   - Toolchain
4. Conclusion & Ongoing Work
### 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

1. Premises

2. Background

3. ReSpecT\textsuperscript{X}: eXtended ReSpecT
   - Modularity, Composability
   - Toolchain

4. Conclusion & Ongoing Work
Modularity $\Rightarrow$ Re-usability, and Composability

- A module definition contains an arbitrary number of:
  - include $\langle$QualifiedName$\rangle$, which imports reactions defined in the referenced module
  - Prolog facts and rules
  - ReSpecT \text{X} reactions

- A specification does the same, but is also translated by the ReSpecT \text{X} compiler in a plain ReSpecT program (directly executable by TuCSoN)

- Reactions can be decorated with a @$\langle$ReactionName$\rangle$ tag
  $\Rightarrow$ referenceable by meta-coordination primitives (primitives with a ReSpecT \text{X} specification tuple as argument)

- Tagged reactions can be further decorated with keyword virtual
  $\Rightarrow$ inactive until the meta-coordination primitive activates them
Example: Scheduling Periodic Activities I

The module outputs the \( A \) tuple \textit{once every} \( P \) milliseconds \( \Rightarrow \) the activity represented by a reaction with \( \text{out}(A) \) as triggering event gets executed periodically.

- ✓ the module can be imported and \textit{reused at will}
- ✓ tagged, virtual reaction exploited
Example: Scheduling Periodic Activities II

```plaintext
module rsp.timing.Periodic {
  reaction out startP(P, A) : completion {
    current_time(Now)
    out context(P, 0, Now, A),
    out tick(A)
  }

  reaction out tick(Activity) : endo, ?context(P, _, _, A) {
    current_time(Now),
    NextTick is Now + P,
    out s @next_tick(NextTick, A)
  }

  @next_tick(T, A)
  virtual reaction time(T) : ??context(_, Tick, _, A), ??tick(A) {
    NextTick is Tick + 1,
    current_time(Now),
    out context(P, NextTick, Now, A),
    out A,
    out tick(A)
  }
}
```
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

module rsp.lang.SetBehaviour {
    put_one(Tuple) :-
        if nop set(Tuple, _) then out set(Tuple, 1)
        else if inp set(Tuple, M) then (  
            NextM is M + 1,
            out set(Tuple, NextM),
            if (NextM > 1) then inp set(Tuple)
        ) else fail.
    reaction out set(Tuple) : completion, exo { put_one(Tuple) }
    remove_one(Tuple) :-
        if inp set(Tuple, M) then (  
            if (M > 0) then (  
                NextM is M - 1,
                out set(Tuple, NextM),
                in_all set(Tuple) returns _,
                if (NextM > 0) then out set(Tuple)
            )
        )  
        reaction inp set(Tuple) : completion, exo { remove_one(Tuple) }  
}
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

```
module rsp.lang.Decay {
    include rsp.lang.Concentration
    include rsp.timing.Periodic

    decay_one(Something) :-
        if (Something = set(Tuple)) then (remove_one(Tuple))
        else (inp Something).
        @decay
        reaction out decay(Something) {
            inp decay(Something),
            decay_one(Something)
        }

    % Remember that startP(P, decay(TT)) tuple
    % triggers periodic emission of decay(TT)
}
```
ReSpecTX standard library\textsuperscript{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]

\textsuperscript{a}http://bitbucket.org/gciatto/respectx-standard-library
Outline

1. Premises
2. Background
3. ReSpecT\textsuperscript{X}: e\textsuperscript{X}tended ReSpecT
   - Modularity, Composability
   - Toolchain
4. Conclusion & Ongoing Work
Toolchain: Static-checking, Code Completion, Code Generation I

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

<table>
<thead>
<tr>
<th>invocation, completion</th>
<th>endo, exo</th>
</tr>
</thead>
<tbody>
<tr>
<td>intra, inter</td>
<td>success, failure</td>
</tr>
<tr>
<td>from_agent, from_tc</td>
<td>to_agent, to_agent</td>
</tr>
<tr>
<td>?X, !Y if X = Y, ground(X)</td>
<td>before(T1), after(T2) if T1 $\geq$ T2</td>
</tr>
</tbody>
</table>

---

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

3 [http://eclipse.org/Xtext/](http://eclipse.org/Xtext/)
Outline

1. Premises

2. Background

3. ReSpecTX: eXtended ReSpecT
   - Modularity, Composability
   - Toolchain

4. Conclusion & Ongoing Work
Conclusion & Ongoing Work

- ReSpecTX 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 ReSpecTX maturity include
  - development of a *rich standard library* of ready-to-use composable coordination mechanisms
  - distribution of ReSpecTX as *ready-to-install* Eclipse IDE plugin
  - *improve static checker*
Premises

Background

ReSpecTX: extended ReSpecT
- Modularity, Composability
- Toolchain

Conclusion & Ongoing Work
Reo: A channel-based coordination model for component composition.

*Developing Multi-Agent Systems with JADE*.  
Wiley.

*Programming Multi-Agent Systems in AgentSpeak using Jason*.  
John Wiley & Sons, Ltd.

Description and composition of bio-inspired design patterns: a complete overview.

Generative communication in Linda.

Mariani, S. and Omicini, A. (2016).  
Multi-paradigm coordination for MAS: Integrating heterogeneous coordination approaches in MAS technologies.  

From tuple spaces to tuple centres.

Coordination models and languages: From parallel computing to self-organisation.

Coordination for Internet application development.

Coordination as a service.  
*Fundamenta Informaticae, 73*(4):507–534.  
Special Issue: Best papers of FOCLASA 2002.

Why interaction is more powerful than algorithms.
Programming the Interaction Space Effectively with ReSpecTX

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

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

School of Electrical Engineering
Belgrade, October 12th 2017