Coordination-based Systems

Distributed Systems
Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Ingegneria Due
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2011/2012

Outline

Outline

Scenarios for Concurrent / Distributed Systems

Issues

- Concurrency / Parallelism
  - Multiple independent activities / loci of control
  - Active simultaneously
  - Processes, threads, actors, active objects, agents...
- Distribution
  - Activities running on different and heterogeneous execution contexts (machines, devices, ...)
- "Social" Interaction
  - Dependencies among activities
  - Collective goals involving activities coordination / cooperation
- "Environmental" Interaction
  - Interaction with external resources
  - Interaction within the time-space fabric

Scenarios for Concurrent / Distributed Systems

Issues

- Concurrency / Parallelism
  - Multiple independent activities / loci of control
  - Active simultaneously
  - Processes, threads, actors, active objects, agents. . .
- Distribution
  - Activities running on different and heterogeneous execution contexts (machines, devices, . . .)
- "Social" Interaction
  - Dependencies among activities
  - Collective goals involving activities coordination / cooperation
- "Environmental" Interaction
  - Interaction with external resources
  - Interaction within the time-space fabric

Scenarios for Concurrent / Distributed Systems

Issues

- Concurrency / Parallelism
  - Multiple independent activities / loci of control
  - Active simultaneously
  - Processes, threads, actors, active objects, agents...
- Distribution
  - Activities running on different and heterogeneous execution contexts (machines, devices, ...)
- "Social" Interaction
  - Dependencies among activities
  - Collective goals involving activities coordination / cooperation
- "Environmental" Interaction
  - Interaction with external resources
  - Interaction within the time-space fabric

Scenarios for Concurrent / Distributed Systems

Issues

- Concurrency / Parallelism
  - Multiple independent activities / loci of control
  - Active simultaneously
  - Processes, threads, actors, active objects, agents...
- Distribution
  - Activities running on different and heterogeneous execution contexts (machines, devices, ...)
- "Social" Interaction
  - Dependencies among activities
  - Collective goals involving activities coordination / cooperation
- "Environmental" Interaction
  - Interaction with external resources
  - Interaction within the time-space fabric

Scenarios for Concurrent / Distributed Systems

Issues

- Concurrency / Parallelism
  - Multiple independent activities / loci of control
  - Active simultaneously
  - Processes, threads, actors, active objects, agents...
- Distribution
  - Activities running on different and heterogeneous execution contexts (machines, devices, ...)
- "Social" Interaction
  - Dependencies among activities
  - Collective goals involving activities coordination / cooperation
- "Environmental" Interaction
  - Interaction with external resources
  - Interaction within the time-space fabric

Basic Engineering Principles

Principles

- Abstraction
  - Problems should be faced / represented at the most suitable level of abstraction
  - Resulting "abstractions" should be expressive enough to capture the most relevant problems
  - Conceptual integrity
- Locality & encapsulation
  - Design abstractions should embody the solutions corresponding to the domain entities they represent
- Run-time vs. design-time abstractions
  - Incremental change / evolutions
  - On-line engineering [Fredriksson and Gustavsson, 2004]
  - (Cognitive) Self-organising systems

Basic Engineering Principles

Principles

- Abstraction
  - Problems should be faced / represented at the most suitable level of abstraction
  - Resulting "abstractions" should be expressive enough to capture the most relevant problems
  - Conceptual integrity
- Locality & encapsulation
  - Design abstractions should embody the solutions corresponding to the domain entities they represent
- Run-time vs. design-time abstractions
  - Incremental change / evolutions
  - On-line engineering [Fredriksson and Gustavsson, 2004]
  - (Cognitive) Self-organising systems

Basic Engineering Principles

Principles

- Abstraction
  - Problems should be faced / represented at the most suitable level of abstraction
  - Resulting "abstractions" should be expressive enough to capture the most relevant problems
  - Conceptual integrity
- Locality & encapsulation
  - Design abstractions should embody the solutions corresponding to the domain entities they represent
- Run-time vs. design-time abstractions
  - Incremental change / evolutions
  - On-line engineering [Fredriksson and Gustavsson, 2004]
  - (Cognitive) Self-organising systems

Basic Engineering Principles

Principles

- Abstraction
  - Problems should be faced / represented at the most suitable level of abstraction
  - Resulting "abstractions" should be expressive enough to capture the most relevant problems
  - Conceptual integrity
- Locality & encapsulation
  - Design abstractions should embody the solutions corresponding to the domain entities they represent
- Run-time vs. design-time abstractions
  - Incremental change / evolutions
  - On-line engineering [Fredriksson and Gustavsson, 2004]
  - (Cognitive) Self-organising systems

Which Components?

Open systems

- No hypothesis on the component's life & behaviour

Distributed systems

- No hypothesis on the component's location & motion

Heterogeneous systems

- No hypothesis on the component's nature & structure

Which Components?

Open systems

- No hypothesis on the component's life & behaviour

Distributed systems

- No hypothesis on the component's location & motion

Heterogeneous systems

- No hypothesis on the component's nature & structure

Which Components?

Open systems

- No hypothesis on the component's life & behaviour

Distributed systems

- No hypothesis on the component's location & motion

Heterogeneous systems

- No hypothesis on the component's nature & structure

Which Interaction? Control vs. Data

How to model an independent activity?

- Objects? No way
  - Objects encapsulate a state and a behaviour, but not a control flow
    - Objects have autonomy over their state, they can control it
    - Objects have not autonomy over their behaviour, they cannot control it
    - Control flows along with data, by means of method invocation (as a reification of message passing)
  - Control is outside objects, owned by human designer who acts as a control authority, establishing the control flow
  - Object interaction is limited and disciplined by interfaces, governed by the human designer

How to model concurrent activities?

- How to model interaction and coordination among concurrent activities?
- How to decouple data and control?
- Method invocation? No way!

Which Interaction? Control vs. Data

How to model an independent activity?

- Objects? No way
  - Objects encapsulate a state and a behaviour, but not a control flow
    - Objects have autonomy over their state, they can control it
    - Objects have not autonomy over their behaviour, they cannot control it
    - Control flows along with data, by means of method invocation (as a reification of message passing)
  - Control is outside objects, owned by human designer who acts as a control authority, establishing the control flow
  - Object interaction is limited and disciplined by interfaces, governed by the human designer

How to model concurrent activities?

- How to model interaction and coordination among concurrent activities?
- How to decouple data and control?
- Method invocation? No way!

Which Interaction? Control vs. Data

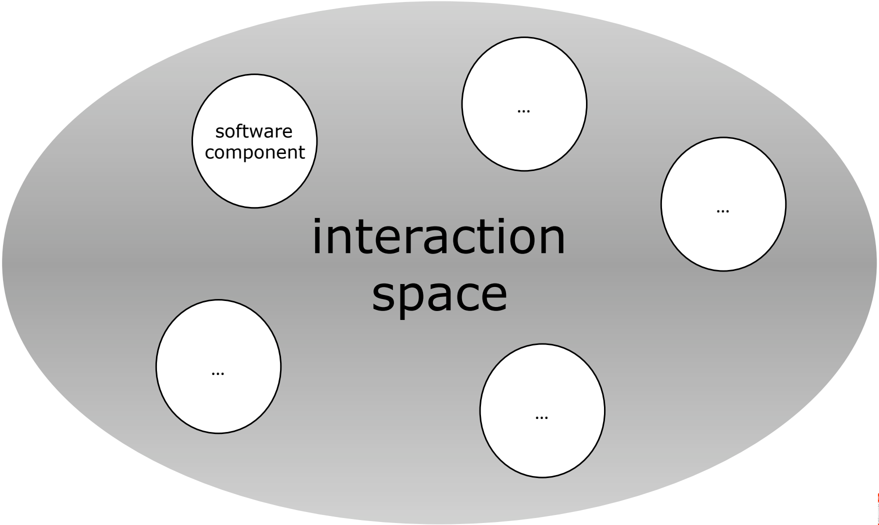
How to model an independent activity?

- Objects? No way
  - Objects encapsulate a state and a behaviour, but not a control flow
    - Objects have autonomy over their state, they can control it
    - Objects have not autonomy over their behaviour, they cannot control it
    - Control flows along with data, by means of method invocation (as a reification of message passing)
  - Control is outside objects, owned by human designer who acts as a control authority, establishing the control flow
  - Object interaction is limited and disciplined by interfaces, governed by the human designer

How to model concurrent activities?

- How to model interaction and coordination among concurrent activities?
- How to decouple data and control?
- Method invocation? No way!

The Space of Interaction



Components of an Interactive System

What is a component of an interactive system?

- A computational abstraction characterised by an independent computational activity, and by I/O capabilities
- Independent elaboration / computation and interaction

Algorithmic Computation

Elaboration / Computation

- Turing Machine
- Black box algorithms
- Church and computable functions

Beyond Turing Machines

- Wegner's Interaction Machines [Goldin et al., 2006]
- Examples: AGV, Chess oracle

Algorithmic Computation

Elaboration / Computation

- Turing Machine
- Black box algorithms
- Church and computable functions

Beyond Turing Machines

- Wegner's Interaction Machines [Goldin et al., 2006]
- Examples: AGV, Chess oracle

Basics of Interaction

A simple sequential machine

- Output: shows part of its state outside
- Input: bounds a portion of its own state to the outside

Coupling across component's boundaries

- Information
- Time – internal / sequential vs. external / entropic

Basics of Interaction

A simple sequential machine

- Output: shows part of its state outside
- Input: bounds a portion of its own state to the outside

Coupling across component's boundaries

- Information
- Time – internal / sequential vs. external / entropic

Compositionality vs. Non-compositionality

Compositionality

- Sequential composition $P1; P2$
- $behaviour(P1; P2) = behaviour(P1) + behaviour(P2)$

Non-compositionality

- Interactive composition $P1|P2$
- $behaviour(P1|P2) =$
  $behaviour(P1) + behaviour(P2) + \textbf{interaction}(P1, P2)$
- Interactive composition is more than the sum of its parts

Compositionality vs. Non-compositionality

Compositionality

- Sequential composition $P1; P2$
- $behaviour(P1; P2) = behaviour(P1) + behaviour(P2)$

Non-compositionality

- Interactive composition $P1|P2$
- $behaviour(P1|P2) =$
  $behaviour(P1) + behaviour(P2) + \mathbf{interaction}(P1, P2)$
- Interactive composition is more than the sum of its parts

Non-compositionality

Issues

- Compositionality vs. formalisability
  - A notion of formal model is required for stating any compositional property
  - However, formalisability does not require compositionality, and does not imply predictability
  - *Partial formalisability* may allow for proof of properties, and for partial predictability
- Emergent behaviours
  - Fully-predictabile / formalisable systems do not allow by definition for emergent behaviours
- Formalisability vs. expressiveness
  - Less / more formalisable systems are (respectively) more / less expressive in terms of potential behaviours

Coordination in Distributed Programming

Coordination model as a glue

A coordination model is the glue that binds separate activities into an ensemble [Gelernter and Carriero, 1992]

Coordination model as an agent interaction framework

A coordination model provides a framework in which the interaction of active and independent entities called agents can be expressed [Ciancarini, 1996]

Issues for a coordination model

A coordination model should cover the issues of creation and destruction of agents, communication among agents, and spatial distribution of agents, as well as synchronization and distribution of their actions over time [Ciancarini, 1996]

Coordination in Distributed Programming

Coordination model as a glue

A coordination model is the glue that binds separate activities into an ensemble [Gelernter and Carriero, 1992]

Coordination model as an agent interaction framework

A coordination model provides a framework in which the interaction of active and independent entities called agents can be expressed [Ciancarini, 1996]

Issues for a coordination model

A coordination model should cover the issues of creation and destruction of agents, communication among agents, and spatial distribution of agents, as well as synchronization and distribution of their actions over time [Ciancarini, 1996]

Coordination in Distributed Programming

Coordination model as a glue

A coordination model is the glue that binds separate activities into an ensemble [Gelernter and Carriero, 1992]

Coordination model as an agent interaction framework

A coordination model provides a framework in which the interaction of active and independent entities called agents can be expressed [Ciancarini, 1996]

Issues for a coordination model

A coordination model should cover the issues of creation and destruction of agents, communication among agents, and spatial distribution of agents, as well as synchronization and distribution of their actions over time [Ciancarini, 1996]

Coordination in Distributed Programming

Coordination model as a glue

A coordination model is the glue that binds separate activities into an ensemble [Gelernter and Carriero, 1992]

Coordination model as an agent interaction framework

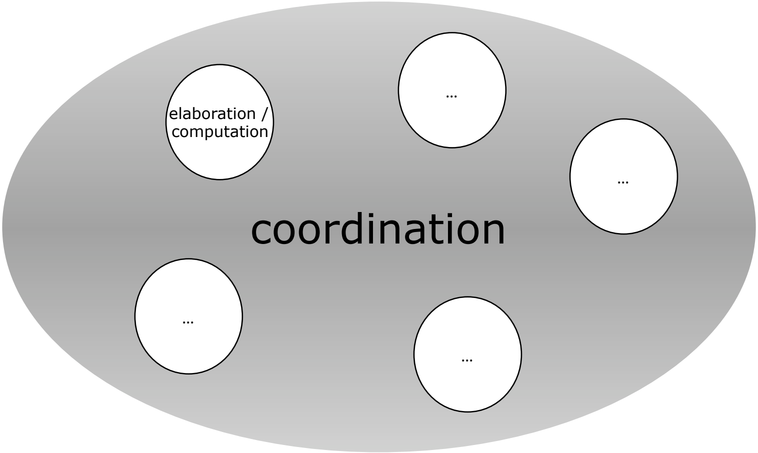
A coordination model provides a framework in which the interaction of active and independent entities called agents can be expressed [Ciancarini, 1996]

Issues for a coordination model

A coordination model should cover the issues of creation and destruction of agents, communication among agents, and spatial distribution of agents, as well as synchronization and distribution of their actions over time [Ciancarini, 1996]

What is Coordination?

Ruling the space of interaction



New Perspective on Computational Systems

Programming languages

- Interaction as an orthogonal dimension
- Languages for interaction / coordination

Software engineering

- Interaction as an independent design dimension
- Coordination patterns

Artificial intelligence

- Interaction as a new source for intelligence
- Social intelligence

New Perspective on Computational Systems

Programming languages

- Interaction as an orthogonal dimension
- Languages for interaction / coordination

Software engineering

- Interaction as an independent design dimension
- Coordination patterns

Artificial intelligence

- Interaction as a new source for intelligence
- Social intelligence

New Perspective on Computational Systems

Programming languages

- Interaction as an orthogonal dimension
- Languages for interaction / coordination

Software engineering

- Interaction as an independent design dimension
- Coordination patterns

Artificial intelligence

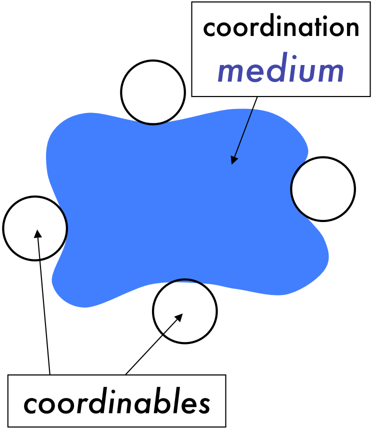
- Interaction as a new source for intelligence
- Social intelligence

Outline

Coordination: Sketching a Meta-model

The *medium of coordination*

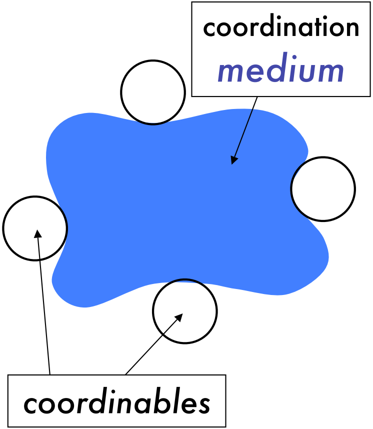
- "fills" the interaction space
- enables / promotes / governs the admissible / desirable / required interactions among the interacting entities
- according to some *coordination laws*
    - enacted by the behaviour of the medium
    - defining the semantics of coordination



Coordination: Sketching a Meta-model

The *medium of coordination*

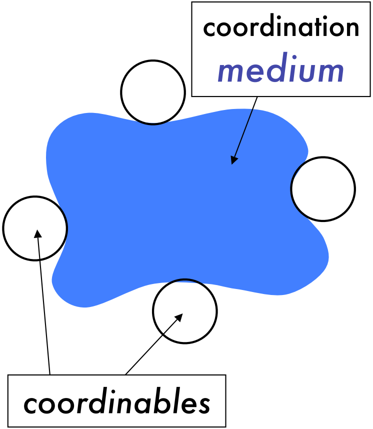
- “fills” the interaction space
- enables / promotes / governs the admissible / desirable / required interactions among the interacting entities
- according to some *coordination laws*
    - enacted by the behaviour of the medium
    - defining the semantics of coordination



Coordination: Sketching a Meta-model

The *medium of coordination*

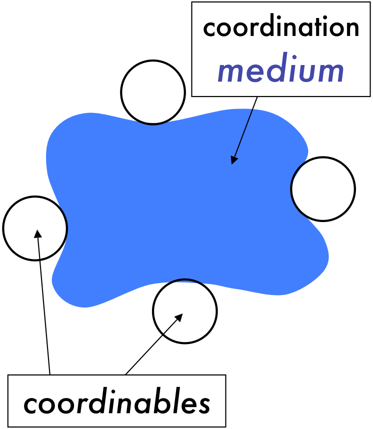
- "fills" the interaction space
- enables / promotes / governs the admissible / desirable / required interactions among the interacting entities
- according to some *coordination laws*
    - enacted by the behaviour of the medium
    - defining the semantics of coordination



Coordination: Sketching a Meta-model

The *medium of coordination*

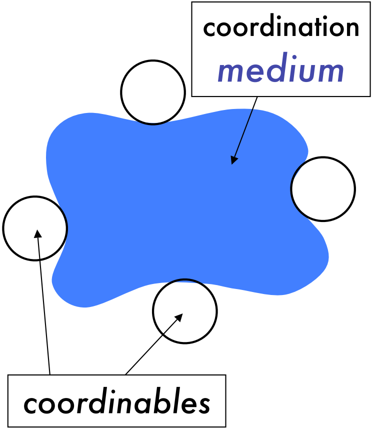
- "fills" the interaction space
- enables / promotes / governs the admissible / desirable / required interactions among the interacting entities
- according to some *coordination laws*
    - enacted by the behaviour of the medium
    - defining the semantics of coordination



Coordination: Sketching a Meta-model

The *medium of coordination*

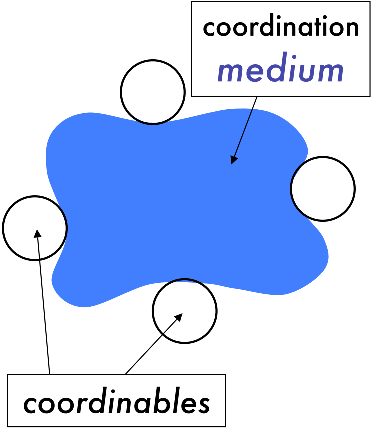
- "fills" the interaction space
- enables / promotes / governs the admissible / desirable / required interactions among the interacting entities
- according to some *coordination laws*
  - enacted by the behaviour of the medium
  - defining the semantics of coordination



Coordination: Sketching a Meta-model

The *medium of coordination*

- "fills" the interaction space
- enables / promotes / governs the admissible / desirable / required interactions among the interacting entities
- according to some *coordination laws*
  - enacted by the behaviour of the medium
  - defining the semantics of coordination



Coordination: A Meta-model [Ciancarini, 1996]

A constructive approach

Which are the components of a coordination system?

Coordination entities
: Entities whose mutual interaction is ruled by the model, also called the *coordinables*

Coordination media
: Abstractions enabling and ruling interaction among coordinables

Coordination laws
: Laws ruling the observable behaviour of coordination media and coordinables, and their interaction as well

Coordination: A Meta-model [Ciancarini, 1996]

A constructive approach

Which are the components of a coordination system?

Coordination entities: Entities whose mutual interaction is ruled by the model, also called the *coordinables*

Coordination media: Abstractions enabling and ruling interaction among coordinables

Coordination laws: Laws ruling the observable behaviour of coordination media and coordinables, and their interaction as well

Coordination: A Meta-model [Ciancarini, 1996]

A constructive approach

Which are the components of a coordination system?

Coordination entities Entities whose mutual interaction is ruled by the model, also called the *coordinables*

Coordination media Abstractions enabling and ruling interaction among coordinables

Coordination laws Laws ruling the observable behaviour of coordination media and coordinables, and their interaction as well

Coordination: A Meta-model [Ciancarini, 1996]

A constructive approach

Which are the components of a coordination system?

Coordination entities Entities whose mutual interaction is ruled by the model, also called the *coordinables*

Coordination media Abstractions enabling and ruling interaction among coordinables

Coordination laws Laws ruling the observable behaviour of coordination media and coordinables, and their interaction as well

Coordination: A Meta-model [Ciancarini, 1996]

A constructive approach

Which are the components of a coordination system?

Coordination entities Entities whose mutual interaction is ruled by the model, also called the *coordinables*

Coordination media Abstractions enabling and ruling interaction among coordinables

Coordination laws Laws ruling the observable behaviour of coordination media and coordinables, and their interaction as well

Coordinables

Original definition [Ciancarini, 1996]

These are the entity types that are coordinated. These could be Unix-like processes, threads, concurrent objects and the like, and even users.

examples Processes, threads, objects, human users, agents, ...

focus Observable behaviour of the coordinables

question Are we anyhow concerned here with the internal machinery / functioning of the coordinable, in principle?

→ This issue will be clear when comparing Linda & TuCSoN agents

Coordinables

Original definition [Ciancarini, 1996]

These are the entity types that are coordinated. These could be Unix-like processes, threads, concurrent objects and the like, and even users.

examples Processes, threads, objects, human users, agents, ...

focus Observable behaviour of the coordinables

question Are we anyhow concerned here with the internal machinery / functioning of the coordinable, in principle?

→ This issue will be clear when comparing Linda & TuCSoN agents

Coordinables

Original definition [Ciancarini, 1996]

These are the entity types that are coordinated. These could be Unix-like processes, threads, concurrent objects and the like, and even users.

examples Processes, threads, objects, human users, agents, ...

focus Observable behaviour of the coordinables

question Are we anyhow concerned here with the internal machinery / functioning of the coordinable, in principle?

→ This issue will be clear when comparing Linda & TuCSoN agents

Coordinables

Original definition [Ciancarini, 1996]

These are the entity types that are coordinated. These could be Unix-like processes, threads, concurrent objects and the like, and even users.

- **examples** Processes, threads, objects, human users, agents, ...
- **focus** Observable behaviour of the coordinables
- **question** Are we anyhow concerned here with the internal machinery / functioning of the coordinable, in principle?

→ This issue will be clear when comparing Linda & TuCSoN agents

Coordinables

Original definition [Ciancarini, 1996]

These are the entity types that are coordinated. These could be Unix-like processes, threads, concurrent objects and the like, and even users.

examples Processes, threads, objects, human users, agents, ...

focus Observable behaviour of the coordinables

question Are we anyhow concerned here with the internal machinery / functioning of the coordinable, in principle?

$\rightarrow$ This issue will be clear when comparing Linda & TuCSoN agents

Coordination media

Original definition [Ciancarini, 1996]

These are the media making communication among the agents possible. Moreover, a coordination medium can serve to aggregate agents that should be manipulated as a whole. Examples are classic media such as semaphores, monitors, or channels, or more complex media such as tuple spaces, blackboards, pipelines, and the like.

- examples: Semaphors, monitors, channels, tuple spaces, blackboards, pipes, . . .
- focus: The core around which the components of the system are organised
- question: Which are the possible computational models for coordination media?
  - → This issue will be clear when comparing Linda tuple spaces & ReSpecT tuple centres

Coordination media

Original definition [Ciancarini, 1996]

These are the media making communication among the agents possible. Moreover, a coordination medium can serve to aggregate agents that should be manipulated as a whole. Examples are classic media such as semaphores, monitors, or channels, or more complex media such as tuple spaces, blackboards, pipelines, and the like.

examples Semaphors, monitors, channels, tuple spaces, blackboards, pipes, . . .

focus The core around which the components of the system are organised

question Which are the possible computational models for coordination media?

→ This issue will be clear when comparing Linda tuple spaces & ReSpecT tuple centres

Coordination media

Original definition [Ciancarini, 1996]

These are the media making communication among the agents possible. Moreover, a coordination medium can serve to aggregate agents that should be manipulated as a whole. Examples are classic media such as semaphores, monitors, or channels, or more complex media such as tuple spaces, blackboards, pipelines, and the like.

examples Semaphors, monitors, channels, tuple spaces, blackboards, pipes, ...

focus The core around which the components of the system are organised

question Which are the possible computational models for coordination media?

→ This issue will be clear when comparing Linda tuple spaces & ReSpecT tuple centres

Coordination media

Original definition [Ciancarini, 1996]

These are the media making communication among the agents possible. Moreover, a coordination medium can serve to aggregate agents that should be manipulated as a whole. Examples are classic media such as semaphores, monitors, or channels, or more complex media such as tuple spaces, blackboards, pipelines, and the like.

examples Semaphors, monitors, channels, tuple spaces, blackboards, pipes, ...

focus The core around which the components of the system are organised

question Which are the possible computational models for coordination media?

→ This issue will be clear when comparing Linda tuple spaces & ReSpecT tuple centres

Coordination media

Original definition [Ciancarini, 1996]

These are the media making communication among the agents possible. Moreover, a coordination medium can serve to aggregate agents that should be manipulated as a whole. Examples are classic media such as semaphores, monitors, or channels, or more complex media such as tuple spaces, blackboards, pipelines, and the like.

examples Semaphors, monitors, channels, tuple spaces, blackboards, pipes, . . .

focus The core around which the components of the system are organised

question Which are the possible computational models for coordination media?

→ This issue will be clear when comparing Linda tuple spaces & ReSpecT tuple centres

Coordination laws

Original definition [Ciancarini, 1996]

A coordination model should dictate a number of laws to describe how agents coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws that enact either synchronous or asynchronous behaviors or exploit explicit or implicit naming schemes for coordination entities.

- Coordination laws rule the observable behaviour of coordination media and coordinables, as well as their interaction
  - a notion of (admissible interaction) event is required to define coordination laws
- The interaction events are (also) expressed in terms of
  - the *communication language*, as the syntax used to express and exchange data structures

  examples tuples, XML elements, FOL terms, (Java) objects, ...
  - the *coordination language*, as the set of the asmissible interaction primitives, along with their semantics

  examples *in/out/rd* (Linda), *send/receive* (channels), *push/pull* (pipes), ...

Coordination laws

Original definition [Ciancarini, 1996]

A coordination model should dictate a number of laws to describe how agents coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws that enact either synchronous or asynchronous behaviors or exploit explicit or implicit naming schemes for coordination entities.

- Coordination laws rule the observable behaviour of coordination media and coordinables, as well as their interaction
  - a notion of (admissible interaction) event is required to define coordination laws
- The interaction events are (also) expressed in terms of
  - the *communication language*, as the syntax used to express and exchange data structures

    examples: tuples, XML elements, FOL terms, (Java) objects, ...
  - the *coordination language*, as the set of the asmissible interaction primitives, along with their semantics

    examples: in/out/rd (Linda), send/receive (channels), push/pull (pipes)

Coordination laws

Original definition [Ciancarini, 1996]

A coordination model should dictate a number of laws to describe how agents coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws that enact either synchronous or asynchronous behaviors or exploit explicit or implicit naming schemes for coordination entities.

- Coordination laws rule the observable behaviour of coordination media and coordinables, as well as their interaction
  - a notion of (admissible interaction) event is required to define coordination laws
- The interaction events are (also) expressed in terms of
  - the *communication language*, as the syntax used to express and exchange data structures

    examples: tuples, XML elements, FOL terms, (Java) objects, ...
  - the *coordination language*, as the set of the asmissible interaction primitives, along with their semantics

    examples: *in/out/rd* (Linda), *send/receive* (channels), *push/pull* (pipes), ...

Coordination laws

Original definition [Ciancarini, 1996]

A coordination model should dictate a number of laws to describe how agents coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws that enact either synchronous or asynchronous behaviors or exploit explicit or implicit naming schemes for coordination entities.

- Coordination laws rule the observable behaviour of coordination media and coordinables, as well as their interaction
  - a notion of (admissible interaction) event is required to define coordination laws
- The interaction events are (also) expressed in terms of
  - the *communication language*, as the syntax used to express and exchange data structures

    examples tuples, XML elements, FOL terms, (Java) objects, ...
  - the *coordination language*, as the set of the asmissible interaction primitives, along with their semantics

    examples in/out/rd (Linda), send/receive (channels), push/pull (pipes), ...

Coordination laws

Original definition [Ciancarini, 1996]

A coordination model should dictate a number of laws to describe how agents coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws that enact either synchronous or asynchronous behaviors or exploit explicit or implicit naming schemes for coordination entities.

- Coordination laws rule the observable behaviour of coordination media and coordinables, as well as their interaction
  - a notion of (admissible interaction) event is required to define coordination laws
- The interaction events are (also) expressed in terms of
  - the *communication language*, as the syntax used to express and exchange data structures

    examples tuples, XML elements, FOL terms, (Java) objects, ...
  - the *coordination language*, as the set of the asmissible interaction primitives, along with their semantics

    examples in/out/rd (Linda), send/receive (channels), push/pull (pipes), ...

Coordination laws

Original definition [Ciancarini, 1996]

A coordination model should dictate a number of laws to describe how agents coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws that enact either synchronous or asynchronous behaviors or exploit explicit or implicit naming schemes for coordination entities.

- Coordination laws rule the observable behaviour of coordination media and coordinables, as well as their interaction
  - a notion of (admissible interaction) event is required to define coordination laws
- The interaction events are (also) expressed in terms of
  - the *communication language*, as the syntax used to express and exchange data structures

  examples tuples, XML elements, FOL terms, (Java) objects, ...

  - the *coordination language*, as the set of the asmissible interaction primitives, along with their semantics

  examples in/out/rd (Linda), send/receive (channels), push/pull (pipes), ...

Coordination laws

Original definition [Ciancarini, 1996]

A coordination model should dictate a number of laws to describe how agents coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws that enact either synchronous or asynchronous behaviors or exploit explicit or implicit naming schemes for coordination entities.

- Coordination laws rule the observable behaviour of coordination media and coordinables, as well as their interaction
  - a notion of (admissible interaction) event is required to define coordination laws
- The interaction events are (also) expressed in terms of
  - the *communication language*, as the syntax used to express and exchange data structures

  examples tuples, XML elements, FOL terms, (Java) objects, ...
  - the *coordination language*, as the set of the asmissible interaction primitives, along with their semantics

  examples in/out/rd (Linda), send/receive (channels), push/pull (pipes), ...

Coordination laws

Original definition [Ciancarini, 1996]

A coordination model should dictate a number of laws to describe how agents coordinate themselves through the given coordination media and using a number of coordination primitives. Examples are laws that enact either synchronous or asynchronous behaviors or exploit explicit or implicit naming schemes for coordination entities.

- Coordination laws rule the observable behaviour of coordination media and coordinables, as well as their interaction
  - a notion of (admissible interaction) event is required to define coordination laws
- The interaction events are (also) expressed in terms of
  - the *communication language*, as the syntax used to express and exchange data structures

examples tuples, XML elements, FOL terms, (Java) objects, ...

  - the *coordination language*, as the set of the asmissible interaction primitives, along with their semantics

examples in/out/rd (Linda), send/receive (channels), push/pull (pipes), ...

Outline

Toward a Notion of Coordination Model

What do we ask to a coordination model?

- to provide high-level *abstractions* and powerful *mechanisms* for distributed system engineering
- to enable and promote the construction of *open, distributed, heterogeneous* systems
- to intrinsically *add properties* to systems independently of components
  - e.g. flexibility, control, intelligence, . . .

Examples of Coordination Mechanisms I

Message passing

- communication among peers
- no abstractions apart from message
- no limitations
    - the notion of *protocol* could be added as a coordination abstraction
- no intrinsic model of coordination
- any pattern of coordination can be superimposed – again, protocols

Examples of Coordination Mechanisms II

Agent Communication Languages

- Goal: promote information exchange
- Examples: Arcol, KQML
- Standard: FIPA ACL
- Semantics: ontologies
- *Enabling communication*
  - ACLs *create* the space of inter-agent communication
  - they do not allow to *constrain* it
- No "real" coordination, again, if not with protocols

Examples of Coordination Mechanisms III

Service-Oriented Architectures

- Basic abstraction: service
- Basic pattern: Service request / response
- Several standards
- Very simple pattern of coordination

Examples of Coordination Mechanisms IV

Web Server

- Basic abstraction: resource (REST/ROA)
- Basic pattern: Resource request / representation / response
- Several standards
- Again, a very simple pattern of coordination
- Generally speaking, objects, HTTP, applets, JavaScript with AJAX, user interface
  - a multi-coordinated systems
  - "spaghetti-coordination", no value added from composition
- How can we "fill" the space of interaction to add value to systems?
  - so, how do we get value from coordination?

Examples of Coordination Mechanisms V

Middleware

- Goal: to provide global properties across distributed systems
- Idea: fill the space of interaction with abstractions and shared features
  - interoperability, security, transactionality, ...
- Middleware can contain coordination abstractions
  - but, it can contain anything, so we need to look at *specific* middleware

Examples of Coordination Mechanisms VI

CORBA

- Goal: managing object interaction across a distributed systems in a transparent way
- Key features: ORB, IDL, CORBAServices. . .
- However, no model for coordination
  - just the client-servant pattern
- However, it can provide a shared support for any coordination abstraction or pattern

Enabling vs. Governing Interaction I

Enabling interaction

- ACL, middleware, mediators. . .
- enabling communication
- enabling components interoperation
- no models for coordination of components
  - no rules on what components should (not) say and do at any given moment, depending on what other components say and do, and on what happens inside and outside the system

Enabling vs. Governing Interaction II

Governing interaction

- ruling communication
- providing concepts, abstractions, models, mechanisms for meaningful component integration
- governing mutual component interaction, and environment-component interaction
- in general, a model that does
  - rule what components should (not) say and do at any given moment
  - depending on what other components say and do, and on what happens inside and outside the system

Outline

Two Classes for Coordination Models

Control-oriented vs. Data-oriented Models

— Control-driven vs. Data-driven Models [Papadopoulos and Arbab, 1998]

Control-oriented Focus on the *acts* of communication

Data-oriented Focus on the *information* exchanged during communication

— Several surveys, no time enough here

— Are these really *classes*?

- actually, better to take this as a criterion to observe coordination models, rather than to separate them

Control-oriented Models I

Processes as black boxes

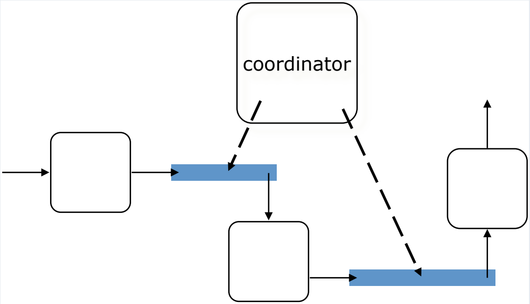
- I/O ports
- events & signals on state

Coordinators. . .

- . . . create coordinated processes as well as communication channels
- . . . determine and change the topology of communication
- Hierarchies of coordinables / coordinators are possible

Control-oriented Models II

Coordinators as meta-level communication components



Control-oriented Models III

General features

- High flexibility, high control
- Separation between communication / coordination and computation / elaboration
- Examples
  - RAPIDE
  - Manifold
  - ConCoord
  - Reo

A Classical Example: Manifold

Main features

- coordinators
- control-driven evolution
    - events without parameters
- stateful communication
- coordination via topology
- fine-grained coordination
- typical example: sort-merge

Control-oriented Models: Impact on Design

Which abstractions?

- Producer-consumer pattern
- Point-to-point communication
- Coordinator
- Coordination as configuration of topology

Which systems?

- Fine-grained granularity
- Fine-tuned control
- Good for small-scale, closed systems

Control-oriented Models: Impact on Design

Which abstractions?

- Producer-consumer pattern
- Point-to-point communication
- Coordinator
- Coordination as configuration of topology

Which systems?

- Fine-grained granularity
- Fine-tuned control
- Good for small-scale, closed systems

An Evolutionary Pattern?

Paradigms of sequential programming

- Imperative programming with "goto"
- Structured programming (procedure-oriented)
- Object-oriented programming (data-oriented)

Paradigms of coordination programming

- Message-passing coordination
- Control-oriented coordination
- Data-oriented coordination

An Evolutionary Pattern?

Paradigms of sequential programming

- Imperative programming with "goto"
- Structured programming (procedure-oriented)
- Object-oriented programming (data-oriented)

Paradigms of coordination programming

- Message-passing coordination
- Control-oriented coordination
- Data-oriented coordination

Data-oriented Models I

Communication channel

- Shared memory abstraction
- Stateful channel

Processes

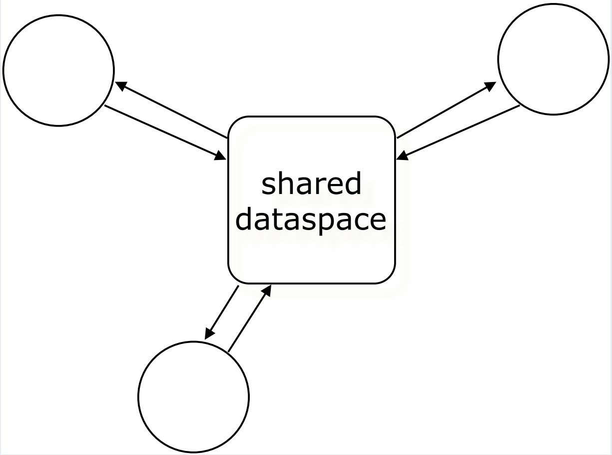
- Emitting / receiving data / information

Coordination

- Access / change / synchronise on shared data

Data-oriented Models II

Shared dataspace: constraint on comunication



Data-oriented Models

General features

- Expressive communication abstraction

→ information-based design

- Possible spatio-temporal uncoupling
- No control means no flexibility??
- Examples
  - Gamma / Chemical coordination
  - Linda & friends / tuple-based coordination

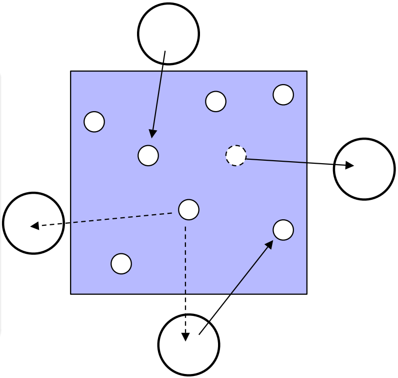
Outline

Outline

The Tuple-space Meta-model

The basics

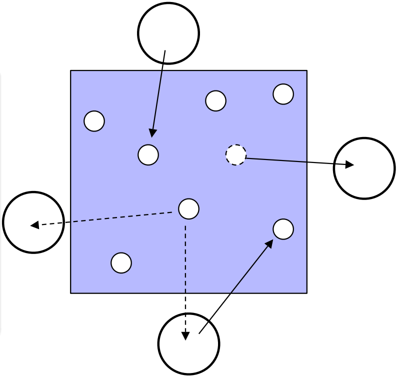
- *Coordinables* synchronise, cooperate, compete
  - based on *tuples*
  - available in the *tuple space*
  - by *associatively* accessing, consuming and producing tuples



The Tuple-space Meta-model

The basics

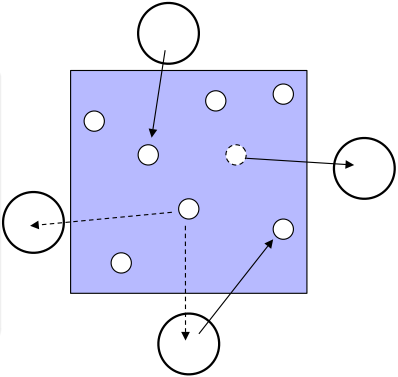
- *Coordinables* synchronise, cooperate, compete
  - based on *tuples*
  - available in the *tuple space*
  - by *associatively* accessing, consuming and producing tuples



The Tuple-space Meta-model

The basics

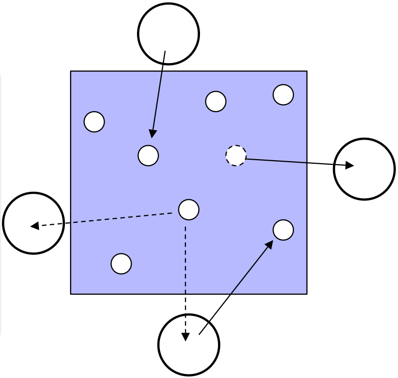
- *Coordinables* synchronise, cooperate, compete
  - based on *tuples*
  - available in the *tuple space*
  - by *associatively* accessing, consuming and producing tuples



The Tuple-space Meta-model

The basics

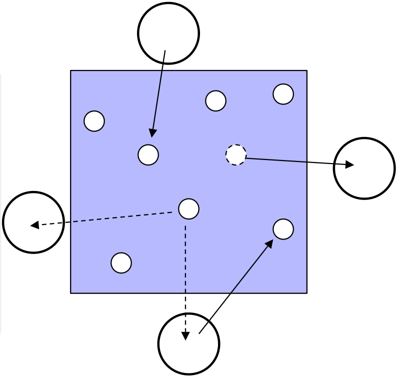
- *Coordinables* synchronise, cooperate, compete
  - based on *tuples*
  - available in the *tuple space*
  - by *associatively* accessing, consuming and producing tuples



The Tuple-space Meta-model

The basics

- *Coordinables* synchronise, cooperate, compete
  - based on *tuples*
  - available in the *tuple space*
  - by *associatively* accessing, consuming and producing tuples



Tuple-based / Space-based Coordination Systems

Adopting the constructive coordination meta-model [Ciancarini, 1996]

coordination media tuple spaces

- as multiset / bag of data objects / structures called *tuples*

communication language tuples

- as ordered collections of (possibly heterogeneous) information items

coordination language tuple space primitives

- as a set of operations to put, browse and retrieve tuples to/from the space

Tuple-based / Space-based Coordination Systems

Adopting the constructive coordination meta-model [Ciancarini, 1996]

coordination media tuple spaces

- as multiset / bag of data objects / structures called *tuples*

communication language tuples

- as ordered collections of (possibly heterogeneous) information items

coordination language tuple space primitives

- as a set of operations to put, browse and retrieve tuples to/from the space

Tuple-based / Space-based Coordination Systems

Adopting the constructive coordination meta-model [Ciancarini, 1996]

coordination media tuple spaces

- as multiset / bag of data objects / structures called *tuples*

communication language tuples

- as ordered collections of (possibly heterogeneous) information items

coordination language tuple space primitives

- as a set of operations to put, browse and retrieve tuples to/from the space

Tuple-based / Space-based Coordination Systems

Adopting the constructive coordination meta-model [Ciancarini, 1996]

coordination media: tuple spaces

- as multiset / bag of data objects / structures called *tuples*

communication language: tuples

- as ordered collections of (possibly heterogeneous) information items

coordination language: tuple space primitives

- as a set of operations to put, browse and retrieve tuples to/from the space

Tuple-based / Space-based Coordination Systems

Adopting the constructive coordination meta-model [Ciancarini, 1996]

coordination media: tuple spaces
- as multiset / bag of data objects / structures called *tuples*

communication language: tuples
- as ordered collections of (possibly heterogeneous) information items

coordination language: tuple space primitives
- as a set of operations to put, browse and retrieve tuples to/from the space

Tuple-based / Space-based Coordination Systems

Adopting the constructive coordination meta-model [Ciancarini, 1996]

coordination media tuple spaces

- as multiset / bag of data objects / structures called *tuples*

communication language tuples

- as ordered collections of (possibly heterogeneous) information items

coordination language tuple space primitives

- as a set of operations to put, browse and retrieve tuples to/from the space

Tuple-based / Space-based Coordination Systems

Adopting the constructive coordination meta-model [Ciancarini, 1996]

coordination media tuple spaces

- as multiset / bag of data objects / structures called *tuples*

communication language tuples

- as ordered collections of (possibly heterogeneous) information items

coordination language tuple space primitives

- as a set of operations to put, browse and retrieve tuples to/from the space

Linda: The Communication Language [Gelernter, 1985]

Communication Language

tuples ordered collections of possibly heterogeneous information chunks

- examples: p(1), printer('HP',dpi(300)), [0,0.5], matrix(m0,3,3,0.5), tree_node(node00,value(13),left(_),right(node01)), ...

templates / anti-tuples specifications of set / classes of tuples

- examples: p(X), [?int,?int], tree_node(N), ...

tuple matching mechanism the mechanism that matches tuples and templates

- examples: pattern matching, unification, ...

Linda: The Communication Language [Gelernter, 1985]

Communication Language

tuples ordered collections of possibly heterogeneous information chunks

- examples: p(1), printer('HP',dpi(300)), [0,0.5], matrix(m0,3,3,0.5), tree_node(node00,value(13),left(_),right(node01)), ...

templates / anti-tuples specifications of set / classes of tuples

- examples: p(X), [?int,?int], tree_node(N), ...

tuple matching mechanism the mechanism that matches tuples and templates

- examples: pattern matching, unification, ...

Linda: The Communication Language [Gelernter, 1985]

Communication Language

tuples ordered collections of possibly heterogeneous information chunks

- examples: p(1), printer('HP',dpi(300)), [0,0.5], matrix(m0,3,3,0.5), tree_node(node00,value(13),left(_),right(node01)), . . .

templates / anti-tuples specifications of set / classes of tuples

- examples: p(X), [?int,?int], tree_node(N), . . .

tuple matching mechanism the mechanism that matches tuples and templates

- examples: pattern matching, unification, . . .

Linda: The Communication Language [Gelernter, 1985]

Communication Language

tuples ordered collections of possibly heterogeneous information chunks

- examples: p(1), printer('HP',dpi(300)), [0,0.5], matrix(m0,3,3,0.5), tree_node(node00,value(13),left(_),right(node01)), ...

templates / anti-tuples specifications of set / classes of tuples

- examples: p(X), [?int,?int], tree_node(N), ...

tuple matching mechanism the mechanism that matches tuples and templates

- examples: pattern matching, unification, ...

Linda: The Communication Language [Gelernter, 1985]

Communication Language

tuples ordered collections of possibly heterogeneous information chunks

- examples: p(1), printer('HP',dpi(300)), [0,0.5], matrix(m0,3,3,0.5), tree_node(node00,value(13),left(_),right(node01)), . . .

templates / anti-tuples specifications of set / classes of tuples

- examples: p(X), [?int,?int], tree_node(N), . . .

tuple matching mechanism the mechanism that matches tuples and templates

- examples: pattern matching, unification, . . .

Linda: The Communication Language [Gelernter, 1985]

Communication Language

tuples ordered collections of possibly heterogeneous information chunks

- examples: p(1), printer('HP',dpi(300)), [0,0.5], matrix(m0,3,3,0.5), tree_node(node00,value(13),left(_),right(node01)), ...

templates / anti-tuples specifications of set / classes of tuples

- examples: p(X), [?int,?int], tree_node(N), ...

tuple matching mechanism the mechanism that matches tuples and templates

- examples: pattern matching, unification, ...

Linda: The Communication Language [Gelernter, 1985]

Communication Language

tuples ordered collections of possibly heterogeneous information chunks

- examples: p(1), printer('HP',dpi(300)), [0,0.5], matrix(m0,3,3,0.5), tree_node(node00,value(13),left(_),right(node01)), ...

templates / anti-tuples specifications of set / classes of tuples

- examples: p(X), [?int,?int], tree_node(N), ...

tuple matching mechanism the mechanism that matches tuples and templates

- examples: pattern matching, unification, ...

Linda: The Coordination Language [Gelernter, 1985] I

out(T)

- out(T) puts tuple T in to the tuple space

  examples `out(p(1))`, `out(0,0.5)`, `out(course('Antonio Natali','Poetry',hours(150))` ...

Linda: The Coordination Language [Gelernter, 1985] II

`in(TT)`

- `in(TT)` retrieves a tuple matching template `TT` from to the tuple space

  destructive reading the tuple retrieved is removed from the tuple centre

  non-determinism if more than one tuple matches the template, one is chosen non-deterministically

  suspensive semantics if no matching tuples are found in the tuple space, operation execution is suspended, and woken when a matching tuple is finally found

  examples `in(p(X))`, `in(0,0.5)`, `in(course('Antonio Natali',Title,hours(X))` ...

Linda: The Coordination Language [Gelernter, 1985] III

rd(TT)

- `rd(TT)` retrieves a tuple matching template `TT` from to the tuple space

  non-destructive reading the tuple retrieved is left untouched in the tuple centre

  non-determinism if more than one tuple matches the template, one is chosen non-deterministically

  suspensive semantics if no matching tuples are found in the tuple space, operation execution is suspended, and awakened when a matching tuple is finally found

  examples `rd(p(X))`, `rd(0,0.5)`, `rd(course('Alessandro Ricci','Operating Systems',hours(X))` ...

Linda Extensions: Predicative Primitives

inp(TT), rdp(TT)

- both `inp(TT)` and `rdp(TT)` retrieve tuple `T` matching template `TT` from the tuple space

= `in(TT)`, `rd(TT)` (non-)destructive reading, non-determinism, and syntax structure is maintained

≠ `in(TT)`, `rd(TT)` suspensive semantics is lost: this *predicative* versions primitives just fail when no tuple matching `TT` is found in the tuple space

success / failure predicative primitives introduce *success / failure semantics*: when a matching tuple is found, it is returned with a success result; when it is not, a failure is reported

Linda Extensions: Predicative Primitives

inp(TT), rdp(TT)

- both `inp(TT)` and `rdp(TT)` retrieve tuple `T` matching template `TT` from the tuple space

= `in(TT)`, `rd(TT)` (non-)destructive reading, non-determinism, and syntax structure is maintained

≠ `in(TT)`, `rd(TT)` suspensive semantics is lost: this *predicative* versions primitives just fail when no tuple matching `TT` is found in the tuple space

success / failure predicative primitives introduce *success* / *failure* *semantics*: when a matching tuple is found, it is returned with a success result; when it is not, a failure is reported

Linda Extensions: Predicative Primitives

inp(TT), rdp(TT)

- both inp(TT) and rdp(TT) retrieve tuple T matching template TT from the tuple space

= in(TT), rd(TT) (non-)destructive reading, non-determinism, and syntax structure is maintained

≠in(TT), rd(TT) suspensive semantics is lost: this *predicative* versions primitives just fail when no tuple matching TT is found in the tuple space

success / failure predicative primitives introduce *success / failure semantics*: when a matching tuple is found, it is returned with a success result; when it is not, a failure is reported

Linda Extensions: Predicative Primitives

inp(TT), rdp(TT)

- both inp(TT) and rdp(TT) retrieve tuple T matching template TT from the tuple space

= in(TT), rd(TT) (non-)destructive reading, non-determinism, and syntax structure is maintained

≠in(TT), rd(TT) suspensive semantics is lost: this *predicative* versions primitives just fail when no tuple matching TT is found in the tuple space

success / failure predicative primitives introduce *success / failure semantics*: when a matching tuple is found, it is returned with a success result; when it is not, a failure is reported

Linda Extensions: Predicative Primitives

`inp(TT)`, `rdp(TT)`

- both `inp(TT)` and `rdp(TT)` retrieve tuple `T` matching template `TT` from the tuple space

  = `in(TT)`, `rd(TT)` (non-)destructive reading, non-determinism, and syntax structure is maintained

  ≠ `in(TT)`, `rd(TT)` suspensive semantics is lost: this *predicative* versions primitives just fail when no tuple matching `TT` is found in the tuple space

  success / failure predicative primitives introduce *success / failure semantics*: when a matching tuple is found, it is returned with a success result; when it is not, a failure is reported

Linda Extensions: Bulk Primitives

in_all(TT), rd_all(TT)

- Linda primitives (including predicative ones) deal with a tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- rd_all(TT), in_all(TT) get all tuples in the tuple space matching with TT, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are rd_all(TT,LT), in_all(TT,LT) (or equivalent), where the (possibly empty) list of tuples unifying with TT is unified with LT
  - (non-)destructive reading: in_all(TT) consumes all matching tuples in the tuple space; rd_all(TT) leaves the tuple space untouched
- Many other bulk primitives have been proposed and implemented to address particular classes of problems

Linda Extensions: Bulk Primitives

`in_all(TT), rd_all(TT)`

- Linda primitives (including predicative ones) deal with a tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- `rd_all(TT)`, `in_all(TT)` get all tuples in the tuple space matching with `TT`, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are `rd_all(TT,LT)`, `in_all(TT,LT)` (or equivalent), where the (possibly empty) list of tuples unifying with `TT` is unified with `LT`
  - (non-)destructive reading: `in_all(TT)` consumes all matching tuples in the tuple space; `rd_all(TT)` leaves the tuple space untouched
- Many other bulk primitives have been proposed and implemented to address particular classes of problems

Linda Extensions: Bulk Primitives

in_all(TT), rd_all(TT)

- Linda primitives (including predicative ones) deal with a tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- rd_all(TT), in_all(TT) get all tuples in the tuple space matching with TT, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are rd_all(TT,LT), in_all(TT,LT) (or equivalent), where the (possibly empty) list of tuples unifying with TT is unified with LT
  - (non-)destructive reading: in_all(TT) consumes all matching tuples in the tuple space; rd_all(TT) leaves the tuple space untouched
- Many other bulk primitives have been proposed and implemented to address particular classes of problems

Linda Extensions: Bulk Primitives

in_all(TT), rd_all(TT)

- Linda primitives (including predicative ones) deal with a tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- rd_all(TT), in_all(TT) get all tuples in the tuple space matching with TT, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are rd_all(TT,LT), in_all(TT,LT) (or equivalent), where the (possibly empty) list of tuples unifying with TT is unified with LT
  - (non-)destructive reading: in_all(TT) consumes all matching tuples in the tuple space; rd_all(TT) leaves the tuple space untouched
- Many other bulk primitives have been proposed and implemented to address particular classes of problems

Linda Extensions: Bulk Primitives

in_all(TT), rd_all(TT)

- Linda primitives (including predicative ones) deal with a tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- rd_all(TT), in_all(TT) get all tuples in the tuple space matching with TT, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are rd_all(TT,LT), in_all(TT,LT) (or equivalent), where the (possibly empty) list of tuples unifying with TT is unified with LT
  - (non-)destructive reading: in_all(TT) consumes all matching tuples in the tuple space; rd_all(TT) leaves the tuple space untouched
- Many other bulk primitives have been proposed and implemented to address particular classes of problems

Linda Extensions: Bulk Primitives

in_all(TT), rd_all(TT)

- Linda primitives (including predicative ones) deal with a tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- rd_all(TT), in_all(TT) get all tuples in the tuple space matching with TT, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are rd_all(TT,LT), in_all(TT,LT) (or equivalent), where the (possibly empty) list of tuples unifying with TT is unified with LT
  - (non-)destructive reading: in_all(TT) consumes all matching tuples in the tuple space; rd_all(TT) leaves the tuple space untouched
- Many other bulk primitives have been proposed and implemented to address particular classes of problems

Linda Extensions: Bulk Primitives

in_all(TT), rd_all(TT)

- Linda primitives (including predicative ones) deal with a tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- rd_all(TT), in_all(TT) get all tuples in the tuple space matching with TT, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are rd_all(TT,LT), in_all(TT,LT) (or equivalent), where the (possibly empty) list of tuples unifying with TT is unified with LT
  - (non-)destructive reading: in_all(TT) consumes all matching tuples in the tuple space; rd_all(TT) leaves the tuple space untouched
- Many other bulk primitives have been proposed and implemented to address particular classes of problems

Linda Extensions: Bulk Primitives

`in_all(TT)`, `rd_all(TT)`

- Linda primitives (including predicative ones) deal with a tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- `rd_all(TT)`, `in_all(TT)` get all tuples in the tuple space matching with `TT`, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are `rd_all(TT,LT)`, `in_all(TT,LT)` (or equivalent), where the (possibly empty) list of tuples unifying with `TT` is unified with `LT`
  - (non-)destructive reading: `in_all(TT)` consumes all matching tuples in the tuple space; `rd_all(TT)` leaves the tuple space untouched
- Many other bulk primitives have been proposed and implemented to address particular classes of problems

Linda Extensions: Bulk Primitives

`in_all(TT)`, `rd_all(TT)`

- Linda primitives (including predicative ones) deal with a tuple at a time
  - some coordination problems require more than one tuple to be handled by a single primitive
- `rd_all(TT)`, `in_all(TT)` get all tuples in the tuple space matching with TT, and returns them all
  - no suspensive semantics: if no matching tuple is found, an empty collection is returned
  - no success / failure semantics: a collection of tuple is always successfully returned—possibly, an empty one
  - in case of logic-based primitives / tuples, the form of the primitive are `rd_all(TT,LT)`, `in_all(TT,LT)` (or equivalent), where the (possibly empty) list of tuples unifying with TT is unified with LT
  - (non-)destructive reading: `in_all(TT)` consumes all matching tuples in the tuple space; `rd_all(TT)` leaves the tuple space untouched
- Many other bulk primitives have been proposed and implemented to address particular classes of problems

Linda Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- Linda tuple space might be a bottleneck for coordination
- Many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- Syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate Linda operators to tuple spaces
- For instance, `ts@node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Linda Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- Linda tuple space might be a bottleneck for coordination
- Many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- Syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate Linda operators to tuple spaces
- For instance, `ts@node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Linda Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- Linda tuple space might be a bottleneck for coordination
- Many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- Syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate Linda operators to tuple spaces
- For instance, `ts@node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Linda Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- Linda tuple space might be a bottleneck for coordination
- Many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- Syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate Linda operators to tuple spaces
- For instance, `ts@node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Linda Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- Linda tuple space might be a bottleneck for coordination
- Many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- Syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate Linda operators to tuple spaces
- For instance, `ts@node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Linda Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- Linda tuple space might be a bottleneck for coordination
- Many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- Syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate Linda operators to tuple spaces
- For instance, `ts@node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Linda Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- Linda tuple space might be a bottleneck for coordination
- Many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- Syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate Linda operators to tuple spaces
- For instance, `ts@node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Linda Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- Linda tuple space might be a bottleneck for coordination
- Many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- Syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate Linda operators to tuple spaces
- For instance, `ts@node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Linda Extensions: Multiple Tuple Spaces

`ts ? out(T)`

- Linda tuple space might be a bottleneck for coordination
- Many extensions have focussed on making a multiplicity of tuple spaces available to processes
  - each of them encapsulating a portion of the coordination load
  - either hosted by a single machine, or distributed across the network
- Syntax required, and dependent on particular models and implementations
  - a space for tuple space names, possibly including network location
  - operators to associate Linda operators to tuple spaces
- For instance, `ts@node ? out(p)` may denote the invocation of operation `out(p)` over tuple space `ts` on node `node`

Main Features of Tuple-based Coordination

Main features of the Linda model

tuples
: A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort

generative communication
: until explicitly withdrawn, the tuples generated by coordinables have an independent existence in the tuple space; a tuple is equally accessible to all the coordinables, but is bound to none

associative access
: tuples in the tuple space are accessed through their content & structure, rather than by name, address, or location

suspensive semantics
: operations may be suspended based on unavailability of matching tuples, and be woken up when such tuples become available

Main Features of Tuple-based Coordination

Main features of the Linda model

tuples A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort

generative communication until explicitly withdrawn, the tuples generated by coordinables have an independent existence in the tuple space; a tuple is equally accessible to all the coordinables, but is bound to none

associative access tuples in the tuple space are accessed through their content & structure, rather than by name, address, or location

suspensive semantics operations may be suspended based on unavailability of matching tuples, and be woken up when such tuples become available

Main Features of Tuple-based Coordination

Main features of the Linda model

tuples A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort

generative communication until explicitly withdrawn, the tuples generated by coordinables have an independent existence in the tuple space; a tuple is equally accessible to all the coordinables, but is bound to none

associative access tuples in the tuple space are accessed through their content & structure, rather than by name, address, or location

suspensive semantics operations may be suspended based on unavailability of matching tuples, and be woken up when such tuples become available

Main Features of Tuple-based Coordination

Main features of the Linda model

tuples A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort

generative communication until explicitly withdrawn, the tuples generated by coordinables have an independent existence in the tuple space; a tuple is equally accessible to all the coordinables, but is bound to none

associative access tuples in the tuple space are accessed through their content & structure, rather than by name, address, or location

suspensive semantics operations may be suspended based on unavailability of matching tuples, and be woken up when such tuples become available

Main Features of Tuple-based Coordination

Main features of the Linda model

tuples: A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort

generative communication: until explicitly withdrawn, the tuples generated by coordinables have an independent existence in the tuple space; a tuple is equally accessible to all the coordinables, but is bound to none

associative access: tuples in the tuple space are accessed through their content & structure, rather than by name, address, or location

suspensive semantics: operations may be suspended based on unavailability of matching tuples, and be woken up when such tuples become available

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Tuples

- A tuple is an ordered collection of knowledge chunks, possibly heterogeneous in sort
  - a record-like structure
  - with no need of field names
  - easy aggregation of knowledge
  - raw semantic interpretation: a tuple contains all information concerning an given item
- Tuple structure based on
  - arity
  - type
  - position
  - information content
- Anti-tuples / Tuple templates
  - to describe / define sets of tuples
- Matching mechanism
  - to define belongingness to a set

Features of Linda: Generative Communication

Communication orthogonality

- Both senders and the receivers can interact even without having prior knowledge about each others

space uncoupling no need to coexist in space for two processes to interact

time uncoupling no need for simultaneity for two processes to interact

name uncoupling no need for names for processes to interact

Features of Linda: Generative Communication

Communication orthogonality

- Both senders and the receivers can interact even without having prior knowledge about each others

space uncoupling: no need to coexist in space for two processes to interact

time uncoupling: no need for simultaneity for two processes to interact

name uncoupling: no need for names for processes to interact

Features of Linda: Generative Communication

Communication orthogonality

- Both senders and the receivers can interact even without having prior knowledge about each others

space uncoupling
: no need to coexist in space for two processes to interact

time uncoupling
: no need for simultaneity for two processes to interact

name uncoupling
: no need for names for processes to interact

Features of Linda: Generative Communication

Communication orthogonality

- Both senders and the receivers can interact even without having prior knowledge about each others

  space uncoupling no need to coexist in space for two processes to interact

  time uncoupling no need for simultaneity for two processes to interact

  name uncoupling no need for names for processes to interact

Features of Linda: Associative Access

Content-based coordination

- Synchronisation based on tuple content & structure
  - absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
  - based on tuple templates & matching mechanism
- *Information-driven coordination*
  - patterns of coordination based on data / information availability
  - based on tuple templates & matching mechanism
- *Reification*
  - making events become tuples
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Associative Access

Content-based coordination

- Synchronisation based on tuple content & structure
  - absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
  - based on tuple templates & matching mechanism
- *Information-driven coordination*
  - patterns of coordination based on data / information availability
  - based on tuple templates & matching mechanism
- *Reification*
  - making events become tuples
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Associative Access

Content-based coordination

- Synchronisation based on tuple content & structure
  - absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
  - based on tuple templates & matching mechanism
- *Information-driven coordination*
  - patterns of coordination based on data / information availability
  - based on tuple templates & matching mechanism
- *Reification*
  - making events become tuples
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Associative Access

Content-based coordination

- Synchronisation based on tuple content & structure
  - absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
  - based on tuple templates & matching mechanism
- *Information-driven coordination*
  - patterns of coordination based on data / information availability
  - based on tuple templates & matching mechanism
- *Reification*
  - making events become tuples
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Associative Access

Content-based coordination

- Synchronisation based on tuple content & structure
  - absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
  - based on tuple templates & matching mechanism
- *Information-driven coordination*
  - patterns of coordination based on data / information availability
  - based on tuple templates & matching mechanism
- *Reification*
  - making events become tuples
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Associative Access

Content-based coordination

- Synchronisation based on tuple content & structure
  - absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
  - based on tuple templates & matching mechanism
- *Information-driven coordination*
  - patterns of coordination based on data / information availability
  - based on tuple templates & matching mechanism
- *Reification*
  - making events become tuples
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Associative Access

Content-based coordination

- Synchronisation based on tuple content & structure
  - absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
  - based on tuple templates & matching mechanism
- *Information-driven coordination*
  - patterns of coordination based on data / information availability
  - based on tuple templates & matching mechanism
- *Reification*
  - making events become tuples
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Associative Access

Content-based coordination

- Synchronisation based on tuple content & structure
  - absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
  - based on tuple templates & matching mechanism
- *Information-driven coordination*
  - patterns of coordination based on data / information availability
  - based on tuple templates & matching mechanism
- *Reification*
  - making events become tuples
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Associative Access

Content-based coordination

- Synchronisation based on tuple content & structure
  - absence / presence of tuples with some content / structure determines the overall behaviour of the coordinables, and of the coordinated system in the overall
  - based on tuple templates & matching mechanism
- *Information-driven coordination*
  - patterns of coordination based on data / information availability
  - based on tuple templates & matching mechanism
- *Reification*
  - making events become tuples
  - grouping classes of events with tuple syntax, and accessing them via tuple templates

Features of Linda: Suspensive Semantics

Blocking primitives

- `in` & `rd` primitives in Linda have a suspensive semantics
  - the coordination medium makes the primitives waiting in case a matching tuple is not found, and wakes it up when such a tuple is found
  - the coordinable invoking the suspensive primitive is expected to wait for its successful completion
- Twofold wait

in the coordination medium the operation is first (possibly) suspended, then (possibly) served: coordination based on absence / presence of tuples belonging to a given set

in the coordination entity the invocation may cause a wait-state in the invoker: hypothesis on the internal behaviour of the coordinable

Features of Linda: Suspensive Semantics

Blocking primitives

- `in` & `rd` primitives in Linda have a suspensive semantics
  - the coordination medium makes the primitives waiting in case a matching tuple is not found, and wakes it up when such a tuple is found
  - the coordinable invoking the suspensive primitive is expected to wait for its successful completion
- Twofold wait

in the coordination medium the operation is first (possibly) suspended, then (possibly) served: coordination based on absence / presence of tuples belonging to a given set

in the coordination entity the invocation may cause a wait-state in the invoker: hypothesis on the internal behaviour of the coordinable

Features of Linda: Suspensive Semantics

Blocking primitives

- `in` & `rd` primitives in Linda have a suspensive semantics
  - the coordination medium makes the primitives waiting in case a matching tuple is not found, and wakes it up when such a tuple is found
  - the coordinable invoking the suspensive primitive is expected to wait for its successful completion
- Twofold wait
  - in the coordination medium: the operation is first (possibly) suspended, then (possibly) served: coordination based on absence / presence of tuples belonging to a given set
  - in the coordination entity: the invocation may cause a wait-state in the invoker: hypothesis on the internal behaviour of the coordinable

Features of Linda: Suspensive Semantics

Blocking primitives

- `in` & `rd` primitives in Linda have a suspensive semantics
  - the coordination medium makes the primitives waiting in case a matching tuple is not found, and wakes it up when such a tuple is found
  - the coordinable invoking the suspensive primitive is expected to wait for its successful completion
- Twofold wait

in the coordination medium the operation is first (possibly) suspended, then (possibly) served: coordination based on absence / presence of tuples belonging to a given set

in the coordination entity the invocation may cause a wait-state in the invoker: hypothesis on the internal behaviour of the coordinable

Features of Linda: Suspensive Semantics

Blocking primitives

- `in` & `rd` primitives in Linda have a suspensive semantics
  - the coordination medium makes the primitives waiting in case a matching tuple is not found, and wakes it up when such a tuple is found
  - the coordinable invoking the suspensive primitive is expected to wait for its successful completion
- Twofold wait

in the coordination medium
: the operation is first (possibly) suspended, then (possibly) served: coordination based on absence / presence of tuples belonging to a given set

in the coordination entity
: the invocation may cause a wait-state in the invoker: hypothesis on the internal behaviour of the coordinable

Features of Linda: Suspensive Semantics

Blocking primitives

- `in` & `rd` primitives in Linda have a suspensive semantics
  - the coordination medium makes the primitives waiting in case a matching tuple is not found, and wakes it up when such a tuple is found
  - the coordinable invoking the suspensive primitive is expected to wait for its successful completion
- Twofold wait

  in the coordination medium the operation is first (possibly) suspended, then (possibly) served: coordination based on absence / presence of tuples belonging to a given set

  in the coordination entity the invocation may cause a wait-state in the invoker: hypothesis on the internal behaviour of the coordinable

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosophers share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosophers share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosophers share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosophers share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosophers share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Our Running Example: The Dining Philosophers Problem

Dining Philosophers [Dijkstra, 2002]

- In the classical Dining Philosopher problem, N philosophers share N chopsticks and a spaghetti bowl
- Each philosopher either eats or thinks
- Each philosopher needs a pair of chopsticks to eat—and can access the two chopsticks on his left and on his right
- Each chopstick is shared by two adjacent philosophers
- When a philosopher needs to think, he gets rid of chopsticks

Concurrency issues in the Dining Philosophers Problem

shared resources Two adjacent philosophers cannot eat simultaneously

starvation If one philosopher eats all the time, the two adjacent philosophers will starve

deadlock If every philosopher picks up the same (say, the left) chopstick at the same time, all of them may wait indefinitely for the other (say, the right) chopstick so as to eat

fairness If a philosopher releases one chopstick before the other one, it favours one of his adjacent philosophers over the other one

Concurrency issues in the Dining Philosophers Problem

shared resources Two adjacent philosophers cannot eat simultaneously

starvation If one philosopher eats all the time, the two adjacent philosophers will starve

deadlock If every philosopher picks up the same (say, the left) chopstick at the same time, all of them may wait indefinitely for the other (say, the right) chopstick so as to eat

fairness If a philosopher releases one chopstick before the other one, it favours one of his adjacent philosophers over the other one

Concurrency issues in the Dining Philosophers Problem

shared resources Two adjacent philosophers cannot eat simultaneously

starvation If one philosopher eats all the time, the two adjacent philosophers will starve

deadlock If every philosopher picks up the same (say, the left) chopstick at the same time, all of them may wait indefinitely for the other (say, the right) chopstick so as to eat

fairness If a philosopher releases one chopstick before the other one, it favours one of his adjacent philosophers over the other one

Concurrency issues in the Dining Philosophers Problem

shared resources Two adjacent philosophers cannot eat simultaneously

starvation If one philosopher eats all the time, the two adjacent philosophers will starve

deadlock If every philosopher picks up the same (say, the left) chopstick at the same time, all of them may wait indefinitely for the other (say, the right) chopstick so as to eat

fairness If a philosopher releases one chopstick before the other one, it favours one of his adjacent philosophers over the other one

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples chop(*i*), that represents the left chopstick for the $i - th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples chop(*i*) chop(*i+1 mod N*)
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples chop(*i*) chop(*i+1 mod N*)

! *In the following, we will use Prolog for philosopher agents*

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(`i`)` `chop(`i`+1 mod N)`

! *In the following, we will use Prolog for philosopher agents*

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(i)`, that represents the left chopstick for the $i - th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i + 1) mod N$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(i) chop(i+1 mod N)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(i) chop(i+1 mod N)`

! *In the following, we will use Prolog for philosopher agents*

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(`*i*`)`, that represents the left chopstick for the $i - th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i + 1) modN$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(`*i*`) chop(`*i+1 mod N*`)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(`*i*`) chop(`*i+1 mod N*`)`

! *In the following, we will use Prolog for philosopher agents*

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(i)`, that represents the left chopstick for the $i - th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) modN$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(i) chop(i+1 mod N)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(i) chop(i+1 mod N)`

! *In the following, we will use Prolog for philosopher agents*

Dining Philosophers in Linda

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by the tuple space
- Chopsticks are represented as tuples `chop(`*i*`)`, that represents the left chopstick for the $i - th$ philosopher
    - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- Philosophers try to eat by getting their chopstick pairs from the tuple space as a pair of tuples `chop(`*i*`)` `chop(`*i+1 mod N*`)`
- Philosophers start to think by releasing their own chopstick pairs to the tuple space as a pair of tuples `chop(`*i*`)` `chop(`*i+1 mod N*`)`

! *In the following, we will use Prolog for philosopher agents*

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly

− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                           % thinking
    in(chop(I)), in(chop(J)),        % waiting to eat
    eat,                             % eating
    out(chop(I)), out(chop(J)),      % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly

− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly

− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                           % thinking
    in(chop(I)), in(chop(J)),        % waiting to eat
    eat,                             % eating
    out(chop(I)), out(chop(J)),      % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly

– starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly

− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                            % thinking
    in(chop(I)), in(chop(J)),         % waiting to eat
    eat,                              % eating
    out(chop(I)), out(chop(J)),       % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly

− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)), in(chop(J)),           % waiting to eat
    eat,                                % eating
    out(chop(I)), out(chop(J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly
− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly
− starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

- + shared resources handled correctly
- − starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly

– starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly
– starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: A Simple Philosopher Protocol

Philosopher using ins and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)), in(chop(J)),       % waiting to eat
    eat,                            % eating
    out(chop(I)), out(chop(J)),     % waiting to think
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly
– starvation, deadlock and unfairness still possible

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                           % thinking
    in(chop(I)),                     % waiting to eat
    (  inp(chop(J)),                 % if other chop available
       eat,                          % eating
       out(chop(I)), out(chop(J)),   % waiting to think
       ;                             % otherwise
       out(chop(I))                  % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                             % thinking
    in(chop(I)),                       % waiting to eat
    (  inp(chop(J)),                   % if other chop available
       eat,                            % eating
       out(chop(I)), out(chop(J)),     % waiting to think
       ;                               % otherwise
       out(chop(I))                    % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (   inp(chop(J)),                   % if other chop available
        eat,                            % eating
        out(chop(I)), out(chop(J)),     % waiting to think
        ;                               % otherwise
        out(chop(I))                    % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                            % thinking
    in(chop(I)),                      % waiting to eat
    (  inp(chop(J)),                  % if other chop available
       eat,                           % eating
       out(chop(I)), out(chop(J)),    % waiting to think
       ;                              % otherwise
       out(chop(I))                   % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
− starvation and unfairness still possible
− not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                % thinking
    in(chop(I)),                          % waiting to eat
    (  inp(chop(J)),                      % if other chop available
       eat,                               % eating
       out(chop(I)), out(chop(J)),        % waiting to think
       ;                                  % otherwise
       out(chop(I))                       % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- – starvation and unfairness still possible
- – not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                  % thinking
    in(chop(I)),                            % waiting to eat
    (   inp(chop(J)),                       % if other chop available
        eat,                                % eating
        out(chop(I)), out(chop(J)),         % waiting to think
        ;                                   % otherwise
        out(chop(I))                        % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                          % thinking
    in(chop(I)),                    % waiting to eat
    (  inp(chop(J)),                % if other chop available
       eat,                         % eating
       out(chop(I)), out(chop(J)),  % waiting to think
       ;                            % otherwise
       out(chop(I))                 % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (  inp(chop(J)),                    % if other chop available
       eat,                             % eating
       out(chop(I)), out(chop(J)),      % waiting to think
       ;                                % otherwise
       out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
− starvation and unfairness still possible
− not-so-trivial philosopher's interaction protocol
    → part of the coordination load is on the coordinables
    → rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                               % thinking
    in(chop(I)),                         % waiting to eat
    (  inp(chop(J)),                     % if other chop available
       eat,                              % eating
       out(chop(I)), out(chop(J)),       % waiting to think
       ;                                 % otherwise
       out(chop(I))                      % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

- \+ shared resources handled correctly, deadlock possibly avoided
- – starvation and unfairness still possible
- – not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                               % thinking
    in(chop(I)),                         % waiting to eat
    (   inp(chop(J)),                    % if other chop available
        eat,                             % eating
        out(chop(I)), out(chop(J)),      % waiting to think
        ;                                % otherwise
        out(chop(I))                     % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (   inp(chop(J)),                   % if other chop available
        eat,                            % eating
        out(chop(I)), out(chop(J)),     % waiting to think
        ;                               % otherwise
        out(chop(I))                    % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                   % thinking
    in(chop(I)),                             % waiting to eat
    (  inp(chop(J)),                         % if other chop available
       eat,                                  % eating
       out(chop(I)), out(chop(J)),           % waiting to think
       ;                                     % otherwise
       out(chop(I))                          % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
  - part of the coordination load is on the coordinables
  - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                 % thinking
    in(chop(I)),                           % waiting to eat
    (   inp(chop(J)),                      % if other chop available
        eat,                               % eating
        out(chop(I)), out(chop(J)),        % waiting to think
        ;                                  % otherwise
        out(chop(I))                       % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                                  % thinking
    in(chop(I)),                            % waiting to eat
    (   inp(chop(J)),                       % if other chop available
        eat,                                % eating
        out(chop(I)), out(chop(J)),         % waiting to think
        ;                                   % otherwise
        out(chop(I))                        % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (   inp(chop(J)),                   % if other chop available
        eat,                            % eating
        out(chop(I)), out(chop(J)),     % waiting to think
        ;                               % otherwise
        out(chop(I))                    % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Another Philosopher Protocol

Philosopher using ins, inps and outs

```
philosopher(I,J) :-
    think,                              % thinking
    in(chop(I)),                        % waiting to eat
    (   inp(chop(J)),                   % if other chop available
        eat,                            % eating
        out(chop(I)), out(chop(J)),     % waiting to think
        ;                               % otherwise
        out(chop(I))                    % releasing unused chop
    )
!,  philosopher(I,J).
```

Issues

+ shared resources handled correctly, deadlock possibly avoided
– starvation and unfairness still possible
– not-so-trivial philosopher's interaction protocol
    - part of the coordination load is on the coordinables
    - rather than on the coordination medium

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                   % thinking
    in(chops(I,J)),          % waiting to eat
    eat,                     % eating
    out(chops(I,J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
− shared resources not handled properly
− starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                    % thinking
    in(chops(I,J)),           % waiting to eat
    eat,                      % eating
    out(chops(I,J)),          % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
− shared resources not handled properly
− starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
− shared resources not handled properly
− starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
– shared resources not handled properly
– starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                   % thinking
    in(chops(I,J)),          % waiting to eat
    eat,                     % eating
    out(chops(I,J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
– shared resources not handled properly
– starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
− shared resources not handled properly
− starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
− shared resources not handled properly
− starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                    % thinking
    in(chops(I,J)),           % waiting to eat
    eat,                      % eating
    out(chops(I,J)),          % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
– shared resources not handled properly
– starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                   % thinking
    in(chops(I,J)),          % waiting to eat
    eat,                     % eating
    out(chops(I,J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
- shared resources not handled properly
- starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
– shared resources not handled properly
– starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                   % thinking
    in(chops(I,J)),          % waiting to eat
    eat,                     % eating
    out(chops(I,J)),         % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
– shared resources not handled properly
– starvation still possible

Dining Philosophers in Linda: Yet Another Philosopher Protocol

Philosopher using ins and outs with chopstick pairs chops(I,J)

```
philosopher(I,J) :-
    think,                  % thinking
    in(chops(I,J)),         % waiting to eat
    eat,                    % eating
    out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

Issues

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
− shared resources not handled properly
− starvation still possible

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Linda: Where is the Problem?

- Coordination is limited to writing, reading, consuming, suspending on one tuple at a time
  - the behaviour of the coordination medium is fixed once and for all
  - coordination problems that fits it are solved satisfactorily, those that do not fit are not
- Bulk primitives are not a general-purpose solution
  - adding ad hoc primitives does not solve the problem in general
  - and does not fit open scenarios—where instead a limited number of well-known primitives are the perfect solution
- As a result, the coordination load is typically charged upon coordination entities
  - this does not fit open scenarios
  - neither it does follow basic software engineering principles, like encapsulation and locality

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Dining Philosophers in Tuple-based Models: Solution?

- Making the behaviour of the coordination medium *adjustable* according to the coordination problem
  - if the behaviour of the coordination medium is *not* be fixed once and for all, and can be defined in accordance to the coordination needs
  - then, in principle all coordination problems may fit some admissible behaviour of the coordination medium
  - with no need to either add new *ad hoc* primitives, or change the semantics of the old ones
- In this way, coordination media could *encapsulate* solutions to coordination problems
  - represented in terms of *coordination policies*
  - enacted in terms of *coordinative behaviour* of the coordination media
- What is needed is a way to *define the behaviour* of a coordination medium according to the specific coordination issues
  - a general *computational model for coordination media*
  - along with a suitably expressive *programming language* to define the behaviour of coordination media

Outline

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N processes have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- In more advanced scenario, these names do not fit
  - *information-driven* coordination vs. *action-driven* coordination fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N processes have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- In more advanced scenario, these names do not fit
  - *information-driven* coordination vs. *action-driven* coordination fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N processes have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- In more advanced scenario, these names do not fit
  - *information-driven* coordination vs. *action-driven* coordination fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N processes have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- In more advanced scenario, these names do not fit
  - *information-driven* coordination vs. *action-driven* coordination fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N processes have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- In more advanced scenario, these names do not fit
  - *information-driven* coordination vs. *action-driven* coordination fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N processes have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- In more advanced scenario, these names do not fit
  - *information-driven* coordination vs. *action-driven* coordination fits better
  - but we might as well use the old terms, while we understand their limitations

Data- vs. Control-driven Coordination

- What if we need to start an activity after, say, at least N processes have asked for a resource?
  - More generally, what if we need, in general, to coordinate based on the coordinable actions, rather than on the information available / exchanged?
- Classical distinction in the coordination community
  - data-driven coordination vs. control-driven coordination
- In more advanced scenario, these names do not fit
  - *information-driven* coordination vs. *action-driven* coordination fits better
  - but we might as well use the old terms, while we understand their limitations

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like Web-based ones, for instance
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of autonomy in distributed systems [Dastani et al., 2005]
  - control should not pass through the component boundaries in order to avoid coupling in distributed systems
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like Web-based ones, for instance
    - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of autonomy in distributed systems [Dastani et al., 2005]
    - control should not pass through the component boundaries in order to avoid coupling in distributed systems
- We need features of both approaches to coordination
    - *hybrid* coordination models
    - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like Web-based ones, for instance
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of autonomy in distributed systems [Dastani et al., 2005]
  - control should not pass through the component boundaries in order to avoid coupling in distributed systems
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like Web-based ones, for instance
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of autonomy in distributed systems [Dastani et al., 2005]
  - control should not pass through the component boundaries in order to avoid coupling in distributed systems
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like Web-based ones, for instance
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of autonomy in distributed systems [Dastani et al., 2005]
  - control should not pass through the component boundaries in order to avoid coupling in distributed systems
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like Web-based ones, for instance
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of autonomy in distributed systems [Dastani et al., 2005]
  - control should not pass through the component boundaries in order to avoid coupling in distributed systems
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Hybrid Coordination Models

- Generally speaking, control-driven coordination does not fit so well information-driven contexts, like Web-based ones, for instance
  - control-driven models like Reo [Arbab, 2004] need to be adapted to agent-based contexts, mainly to deal with the issue of autonomy in distributed systems [Dastani et al., 2005]
  - control should not pass through the component boundaries in order to avoid coupling in distributed systems
- We need features of both approaches to coordination
  - *hybrid* coordination models
  - adding for instance a control-driven layer to a Linda-based one
- What should be added to a tuple-based model to make it hybrid, and how?

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features?
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features?
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features?
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features?
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features?
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features?
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features?
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events

Towards Tuple Centres

- What should be left unchanged?
  - no new primitives
  - basic Linda primitives are preserved, both syntax and semantics
  - matching mechanism preserved, still depending on the communication language of choice
  - multiple tuple spaces, flat name space
- New features?
  - ability to define new coordinative behaviours embodying required coordination policies
  - ability to associate coordinative behaviours to coordination events

Outline

Outline

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable process interaction protocols to be organised around the desired / required process perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, processes could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example. . .

- . . . this would amount to representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while enabling philosophers to perceive chopsticks as pairs (tuples `chops(`i`,`j`)`), so that philosophers could acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i`,`j`))` / `out(chops(`i`,`j`))`.
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
    - in the tuple space
    - this would enable process interaction protocols to be organised around the desired / required process perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
    - so, processes could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to representing each chopstick as a single `chop(i)` tuple in the tuple space, while enabling philosophers to perceive chopsticks as pairs (tuples `chops(i,j)`), so that philosophers could acquire / release two chopsticks by means of a single tuple space operation `in(chops(i,j))` / `out(chops(i,j))`.
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable process interaction protocols to be organised around the desired / required process perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, processes could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example. . .

- . . . this would amount to representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while enabling philosophers to perceive chopsticks as pairs (tuples `chops(`i,j`)`), so that philosophers could acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i,j`))` / `out(chops(`i,j`))`.
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable process interaction protocols to be organised around the desired / required process perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, processes could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example. . .

- . . . this would amount to representing each chopstick as a single `chop(i)` tuple in the tuple space, while enabling philosophers to perceive chopsticks as pairs (tuples `chops(i,j)`), so that philosophers could acquire / release two chopsticks by means of a single tuple space operation `in(chops(i,j))` / `out(chops(i,j))`.
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
    - in the tuple space
    - this would enable process interaction protocols to be organised around the desired / required process perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
    - so, processes could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example. . .

- . . . this would amount to representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while enabling philosophers to perceive chopsticks as pairs (tuples `chops(`i,j`)`), so that philosophers could acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i,j`))` / `out(chops(`i,j`))`.
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable process interaction protocols to be organised around the desired / required process perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, processes could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to representing each chopstick as a single chop(i) tuple in the tuple space, while enabling philosophers to perceive chopsticks as pairs (tuples chops(i,j)), so that philosophers could acquire / release two chopsticks by means of a single tuple space operation in(chops(i,j)) / out(chops(i,j)).
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
   - in the tuple space
   - this would enable process interaction protocols to be organised around the desired / required process perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
   - so, processes could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example...

- ...this would amount to representing each chopstick as a single chop(i) tuple in the tuple space, while enabling philosophers to perceive chopsticks as pairs (tuples chops(i,j)), so that philosophers could acquire / release two chopsticks by means of a single tuple space operation in(chops(i,j)) / out(chops(i,j)).
- How could we do that, in the example, and in general?

Ideas from the Dining Philosophers

1. Keeping information representation and perception separated
    - in the tuple space
    - this would enable process interaction protocols to be organised around the desired / required process perception of the interaction space (tuple space), independently of its *actual* representation in terms of tuples
2. Properly relating information representation and perception through a suitably defined tuple-space behaviour
    - so, processes could get rid of the unnecessary burden of coordination, by embedding coordination laws into the coordination media

In the Dining Philosophers example. . .

- . . . this would amount to representing each chopstick as a single `chop(`i`)` tuple in the tuple space, while enabling philosophers to perceive chopsticks as pairs (tuples `chops(`i,j`)`), so that philosophers could acquire / release two chopsticks by means of a single tuple space operation `in(chops(`i,j`))` / `out(chops(`i,j`))`.
- How could we do that, in the example, and in general?

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- So, in principle, the new tuple-based abstraction should be
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by processes as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing process interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- So, in principle, the new tuple-based abstraction should be
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by processes as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing process interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- So, in principle, the new tuple-based abstraction should be
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by processes as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing process interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- So, in principle, the new tuple-based abstraction should be
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by processes as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing process interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- So, in principle, the new tuple-based abstraction should be
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by processes as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing process interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- So, in principle, the new tuple-based abstraction should be
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by processes as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing process interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- So, in principle, the new tuple-based abstraction should be
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by processes as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing process interaction, a tuple centre may behave in a completely different way with respect to a tuple space

A Possible Solution

- A twofold solution
  1. maintaining the standard tuple space interface
  2. making it possible to enrich the behaviour of a tuple space in terms of the state transitions performed in response to the occurrence of standard communication events
- So, in principle, the new tuple-based abstraction should be
  - a tuple space whose behaviour in response to communication events is no longer fixed once and for all by the coordination model, but can be defined according to the required coordination policies

Consequences

- Since it has exactly the same interface, a tuple centre is perceived by processes as a standard tuple space
- However, since its behaviour can be specified so as to encapsulate the coordination rules governing process interaction, a tuple centre may behave in a completely different way with respect to a tuple space

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Tuple Centres

Definition [Omicini and Denti, 2001]

- A tuple centre is a tuple space enhanced with a *behaviour specification*, defining the behaviour of a tuple centre in response to interaction events
- The *behaviour specification* of tuple centre
  - is expressed in terms of a *reaction specification language*, and
  - associates any tuple-centre event to a (possibly empty) set of computational activities, which are called *reactions*
- More precisely, a reaction specification language
  - enables the definitions of computational activities within a tuple centre, called reactions, and
  - makes it possible to associate reactions to the events that occur in a tuple centre

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing process, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing process, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing process, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing process, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
  - access and modify the current tuple centre state—like adding or removing tuples)
  - access the information related to the triggering event—such as the performing process, the primitive invoked, the tuple involved, etc.)—which is made completely observable
  - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
  - instead, it can be made as complex as required by the specific application needs

Reactions

- Each reaction can in principle
    - access and modify the current tuple centre state—like adding or removing tuples)
    - access the information related to the triggering event—such as the performing process, the primitive invoked, the tuple involved, etc.)—which is made completely observable
    - invoke link primitives upon other tuple centres
- As a result, the semantics of the standard tuple space communication primitives is no longer constrained to be as simple as in the Linda model—i.e., adding, reading, and removing tuples
    - instead, it can be made as complex as required by the specific application needs

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the process's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the process's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the process's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the process's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the process's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the process's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the process's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Reaction Execution

- The main cycle of a tuple centre works as follows
  - when a primitive invocation reaches a tuple centre, all the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the primitive is served in the same way as in standard Linda
  - upon completion of the invocation, the corresponding reactions (if any) are triggered, and then executed in a non-deterministic order
  - once all the reactions have been executed, the main cycle of a tuple centre may go on possibly serving another invocation
- As a result, tuple centres exhibit a couple of fundamental features
  - since an empty behaviour specification brings no triggered reactions independently of the invocation, the behaviour of a tuple centre defaults to a tuple space when no behaviour specification is given
  - from the process's viewpoint, the result of the invocation of a tuple centre primitive is the sum of the effects of the primitive itself and of all the reactions it triggers, perceived altogether as a single-step transition of the tuple centre state

Tuple Centre's State vs. Process's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by processes as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by processes
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the process's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the distributed system

Tuple Centre's State vs. Process's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by processes as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by processes
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the process's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the distributed system

Tuple Centre's State vs. Process's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by processes as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by processes
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the process's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the distributed system

Tuple Centre's State vs. Process's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by processes as a single-step transition of the tuple-centre state
  - as in the case of tuple spaces
  - so tuple centres are perceived as tuple spaces by processes
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
  - this makes it possible to decouple the process's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the distributed system

Tuple Centre's State vs. Process's Perception

- Reactions are executed in such a way that the observable behaviour of a tuple centre in response to a communication event is still perceived by processes as a single-step transition of the tuple-centre state
    - as in the case of tuple spaces
    - so tuple centres are perceived as tuple spaces by processes
- Unlike a standard tuple space, whose state transitions are constrained to adding, reading or deleting one single tuple, the perceived transition of a tuple centre state can be made as complex as needed
    - this makes it possible to decouple the process's view of the tuple centre (perceived as a standard tuple space) from the actual state of a tuple centre, and to relate them so as to embed the coordination laws governing the distributed system

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by processes
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by processes
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by processes
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by processes
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by processes
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by processes
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by processes
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Tuple Centres & Hybrid Coordination

- Tuple centres promote a form of hybrid coordination
  - aimed at preserving the advantages of data-driven models
  - while addressing their limitations in terms of control capabilities
- On the one hand, a tuple centre is basically an information-driven coordination medium, which is perceived as such by processes
- On the other hand, a tuple centre also features some capabilities which are typical of action-driven models, like
  - the full observability of events
  - the ability to selectively react to events
  - the ability to implement coordination rules by manipulating the interaction space

Outline

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- A philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`i`,`i`+1 mod N)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`i`,`i`+1 mod N)` invocation

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(`*i*`)`, that represents the left chopstick for the $i - th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i + 1) mod N$ (right)
- A philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`*i*`,`*i+1 mod N*`)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`*i*`,`*i+1 mod N*`)` invocation

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(i)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- A philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(i,i+1 mod N)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(i,i+1 mod N)` invocation

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(`*i*`)`, that represents the left chopstick for the $i - th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i + 1) modN$ (right)
- A philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`*i*`,`*i+1 mod N*`)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`*i*`,`*i+1 mod N*`)` invocation

Dining Philosophers in ReSpecT

- The spaghetti bowl, or, more easily, the table where the bowl and the chopstick are, and the philosophers are seated, are represented by tuple centre `table`
- Chopsticks are represented as tuples `chop(`i`)`, that represents the left chopstick for the $i-th$ philosopher
  - philosopher i needs chopsticks i (left) and $(i+1) mod N$ (right)
- A philosopher tries to eat by getting his chopstick pair from the tuple centre by means of a `in(chops(`*i*`,`*i+1 mod N*`)` invocation
- A philosopher starts to think by releasing his own chopstick pair to the tuple centre by means of a `out(chops(`*i*`,`*i+1 mod N*`)` invocation

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                         % thinking
    table ? in(chops(I,J)),        % waiting to eat
    eat,                           % eating
    table ? out(chops(I,J)),       % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                       % thinking
    table ? in(chops(I,J)),      % waiting to eat
    eat,                         % eating
    table ? out(chops(I,J)),     % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol

? shared resources handled properly?

? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                      % thinking
    table ? in(chops(I,J)),     % waiting to eat
    eat,                        % eating
    table ? out(chops(I,J)),    % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
? shared resources handled properly?
? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                           % thinking
    table ? in(chops(I,J)),          % waiting to eat
    eat,                             % eating
    table ? out(chops(I,J)),         % waiting to think
!,  philosopher(I,J).
```

Results

- \+ fairness, no deadlock
- \+ trivial philosopher's interaction protocol
- ? shared resources handled properly?
- ? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                      % thinking
    table ? in(chops(I,J)),     % waiting to eat
    eat,                        % eating
    table ? out(chops(I,J)),    % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
? shared resources handled properly?
? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                       % thinking
    table ? in(chops(I,J)),      % waiting to eat
    eat,                         % eating
    table ? out(chops(I,J)),     % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
? shared resources handled properly?
? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                        % thinking
    table ? in(chops(I,J)),       % waiting to eat
    eat,                          % eating
    table ? out(chops(I,J)),      % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
? shared resources handled properly?
? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                       % thinking
    table ? in(chops(I,J)),      % waiting to eat
    eat,                         % eating
    table ? out(chops(I,J)),     % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol
? shared resources handled properly?
? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                        % thinking
    table ? in(chops(I,J)),       % waiting to eat
    eat,                          % eating
    table ? out(chops(I,J)),      % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol

? shared resources handled properly?

? starvation still possible?

Dining Philosophers in ReSpecT: Philosopher Protocol

```
philosopher(I,J) :-
    think,                       % thinking
    table ? in(chops(I,J)),      % waiting to eat
    eat,                         % eating
    table ? out(chops(I,J)),     % waiting to think
!,  philosopher(I,J).
```

Results

+ fairness, no deadlock
+ trivial philosopher's interaction protocol

? shared resources handled properly?

? starvation still possible?

Dining Philosophers in ReSpecT: `table` Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
   in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
   out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
   in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
   in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
   rd(required(C,C2)), in(chop(C)), in(chop(C2)),
   out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5’)
   rd(required(C1,C)), in(chop(C1)), in(chop(C)),
   out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: table Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (       % (1)
   in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (        % (2)
   out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (        % (3)
   in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                   % (4)
   in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                           % (5)
   rd(required(C,C2)), in(chop(C)), in(chop(C2)),
   out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                           % (5'
   rd(required(C1,C)), in(chop(C1)), in(chop(C)),
   out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: table Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: table Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: table Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: table Behaviour Specification

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)),  out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)),
    out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)),
    out(chops(C1,C)) )).
```

Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
-	starvation still possible

Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
-	starvation still possible

Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
-	starvation still possible

Dining Philosophers in ReSpecT: Results

Results

protocol no deadlock

protocol fairness

protocol trivial philosopher's interaction protocol

tuple centre shared resources handled properly

- starvation still possible

Dining Philosophers in ReSpecT: Results

Results

- **protocol** no deadlock
- **protocol** fairness
- **protocol** trivial philosopher's interaction protocol
- **tuple centre** shared resources handled properly
- \- starvation still possible

Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
-	starvation still possible

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosophers are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre seat(i,j)
  - seat(i,j) denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick i is represented as a tuple chop(i) in the table tuple centre
- 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, embedded in the tuple centre behaviour
- *N* individual tuple centres (seat(i,j)) + 1 social tuple centre (table) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosophers are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- 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, embedded in the tuple centre behaviour
- N individual tuple centres (`seat(i,j)`) + 1 social tuple centre (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosophers are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- 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, embedded in the tuple centre behaviour
- N individual tuple centres (`seat(i,j)`) + 1 social tuple centre (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosophers are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- 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, embedded in the tuple centre behaviour
- N individual tuple centres (`seat(i,j)`) + 1 social tuple centre (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosophers are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- 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, embedded in the tuple centre behaviour
- N individual tuple centres (`seat(i,j)`) + 1 social tuple centre (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosophers are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- 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, embedded in the tuple centre behaviour
- *N* individual tuple centres (`seat(i,j)`) + 1 social tuple centre (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosophers are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- 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, embedded in the tuple centre behaviour
- N individual tuple centres (`seat(i,j)`) + 1 social tuple centre (`table`) connected in a star network

Distributed Dining Philosophers

Dining Philosophers in a distributed setting

- N philosophers are distributed along the network
  - each philosopher is assigned a seat, represented by the tuple centre `seat(i,j)`
  - `seat(i,j)` denotes that the associated philosopher needs chopstick pair chops(i,j) so as to eat
- each chopstick `i` is represented as a tuple `chop(i)` in the `table` tuple centre
- 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, embedded in the tuple centre behaviour
- *N* individual tuple centres (`seat(i,j)`) + 1 social tuple centre (`table`) connected in a star network

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the table
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of chop resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the table
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of chop resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the table
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the table
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the table
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the table
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of chop resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the table
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the seat

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

Distributed Dining Philosophers: Individual Interaction

Philosopher–seat interaction (*use*)

- four states, represented by tuple `philosopher(_)`
  - `thinking`, `waiting_to_eat`, `eating`, `waiting_to_think`
- determined by
  - the `out(wanna_eat)` / `out(wanna_think)` invocations, expressing the philosopher's intentions
  - the interaction with the `table` tuple centre, expressing the availability of `chop` resources
- tuple `chops(i,j)` only occurs in tuple centre `seat(i,j)` in the `philosopher(eating)` state
- state transitions only occur when they are safe
  - from `waiting_to_think` to `thinking` only when chopsticks are safely back on the `table`
  - from `waiting_to_eat` to `eating` only when chopsticks are actually at the `seat`

ReSpecT code for seat(i, j) tuple centres

```
reaction( out(wanna_eat), (operation, invocation), (          % (1)
    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),             % (2)
    in(wanna_eat)).
reaction( in(chops(C1,C2)), (link_out, completion), (          % (3)
    in(philosopher(waiting_to_eat)), out(philosopher(eating)),
    out(chops(C1,C2)) )).
reaction( out(wanna_think), (operation, invocation), (          % (4)
    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),           % (5)
    in(wanna_think) ).
reaction( out(chops(C1,C2)), (link_out, completion), (         % (6)
    in(philosopher(waiting_to_think)), out(philosopher(thinking)))
```

Distributed Dining Philosophers: Social Interaction

Seat–table interaction (*link*)

- tuple centre `seat(i,j)` requires / returns tuple `chops(i,j)` from / to `table` tuple centre
- 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

Distributed Dining Philosophers: Social Interaction

Seat–table interaction (*link*)

- tuple centre `seat(i,j)` requires / returns tuple `chops(i,j)` from / to `table` tuple centre
- 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

Distributed Dining Philosophers: Social Interaction

Seat–table interaction (*link*)

- tuple centre `seat(i,j)` requires / returns tuple `chops(i,j)` from / to `table` tuple centre
- 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

```
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)) )).
```

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centre (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social tuple centre (`table` tuple centre)
  - the social tuple centre (`table`) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single processes is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible tuple centres as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centre (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social tuple centre (`table` tuple centre)
  - the social tuple centre (`table`) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
  - accessible, represented in a declarative way

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centre (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social tuple centre (`table` tuple centre)
  - the social tuple centre (`table`) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single processes is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible media as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centre (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social tuple centre (`table` tuple centre)
  - the social tuple centre (`table`) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single processes is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible media as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centre (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social tuple centre (`table` tuple centre)
  - the social tuple centre (`table`) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single processes is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible media as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centre (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social tuple centre (`table` tuple centre)
  - the social tuple centre (`table`) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single processes is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible media as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centre (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social tuple centre (`table` tuple centre)
  - the social tuple centre (`table`) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single processes is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible media as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centre (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social tuple centre (`table` tuple centre)
  - the social tuple centre (`table`) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single processes is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible media as far as the portion representing their social interaction is concerned

Distributed Dining Philosophers: Features

- Full separation of concerns
  - philosophers just express their intentions, in terms of simple tuples
  - individual tuple centre (`seat(i,j)` tuple centres) handle individual behaviours and state, and mediate interaction of individuals with social tuple centre (`table` tuple centre)
  - the social tuple centre (`table`) deals with shared resources (`chop` tuples) and ensures global system properties, like fairness and deadlock avoidance
- At any time, one could look at the coordination media, and find exactly the consistent representation of the current distributed state
  - properly distributed, suitably encapsulated
    - the state of shared resources is in the shared distributed abstraction, the state of single processes is into individual local abstractions
  - accessible, represented in a declarative way
    - the state of individual philosophers is exposed through accessible media as far as the portion representing their social interaction is concerned

Outline

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction`(E, G, R)
  - if event Ev occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
    - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
    - if event Ev occurs in the tuple centre,
    - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
    - guard G is true,
    - then reaction $R\theta$ to Ev is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event *Ev* occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E, Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to *Ev* is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event *Ev* occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E, Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to *Ev* is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event *Ev* occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to *Ev* is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event *Ev* occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E, Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to *Ev* is triggered for execution in the tuple centre

ReSpecT Basic Syntax for Reactions

Logic Tuples

- ReSpecT tuple centres adopt logic tuples for both ordinary tuples and specification tuples
- ordinary tuples are simple first-order logic (FOL) facts, written with a Prolog syntax
  - while ordinary logic tuples are typically ground facts, there is nothing to constrain them to be such
- specification tuples are logic tuples of the form `reaction(`E`,`G`,`R`)`
  - if event *Ev* occurs in the tuple centre,
  - which matches event descriptor E such that $\theta = mgu(E,Ev)$, and
  - guard G is true,
  - then reaction $R\theta$ to *Ev* is triggered for execution in the tuple centre

ReSpecT Core Syntax

⟨*TCSpecification*⟩	::=	{⟨*SpecificationTuple*⟩ `.`}
⟨*SpecificationTuple*⟩	::=	`reaction(` ⟨*SimpleTCEvent*⟩ `,` [⟨*Guard*⟩ `,`] ⟨*Reaction*⟩ `)`
⟨*SimpleTCEvent*⟩	::=	⟨*SimpleTCPredicate*⟩ `(` ⟨*Tuple*⟩ `)` | `time(` ⟨*Time*⟩ `)`
⟨*Guard*⟩	::=	⟨*GuardPredicate*⟩ | `(` ⟨*GuardPredicate*⟩ {`,` ⟨*GuardPredicate*⟩} `)`
⟨*Reaction*⟩	::=	⟨*ReactionGoal*⟩ | `(` ⟨*ReactionGoal*⟩ {`,` ⟨*ReactionGoal*⟩} `)`
⟨*ReactionGoal*⟩	::=	⟨*TCPredicate*⟩ `(` ⟨*Tuple*⟩ `)` | ⟨*ObservationPredicate*⟩ `(` ⟨*Tuple*⟩ `)` | ⟨*Computation*⟩ | `(` ⟨*ReactionGoal*⟩ `;` ⟨*ReactionGoal*⟩ `)`
⟨*TCPredicate*⟩	::=	⟨*SimpleTCPredicate*⟩ | ⟨*TCLinkPredicate*⟩
⟨*TCLinkPredicate*⟩	::=	⟨*TCIdentifier*⟩ `?` ⟨*SimpleTCPredicate*⟩
⟨*SimpleTCPredicate*⟩	::=	⟨*TCStatePredicate*⟩ | ⟨*TCForgePredicate*⟩
⟨*TCStatePredicate*⟩	::=	`in` | `inp` | `rd` | `rdp` | `out` | `no` | `get` | `set`
⟨*TCForgePredicate*⟩	::=	⟨*TCStatePredicate*⟩`_s`
⟨*ObservationPredicate*⟩	::=	⟨*EventView*⟩`_`⟨*EventInformation*⟩
⟨*EventView*⟩	::=	`current` | `event` | `start`
⟨*EventInformation*⟩	::=	`predicate` | `tuple` | `source` | `target` | `time`
⟨*GuardPredicate*⟩	::=	`request` | `response` | `success` | `failure` | `endo` | `exo` | `intra` | `inter` | `from_agent` | `to_agent` | `from_tc` | `to_tc` | `before(` ⟨*Time*⟩ `)` | `after(` ⟨*Time*⟩ `)`
⟨*Time*⟩	is	a non-negative integer
⟨*Tuple*⟩	is	Prolog term
⟨*Computation*⟩	is	a Prolog-like goal performing arithmetic / logic computations
⟨*TCIdentifier*⟩	::=	⟨*TCName*⟩ `@` ⟨*NetworkLocation*⟩
⟨*TCName*⟩	is	a Prolog ground term
⟨*NetworkLocation*⟩	is	a Prolog string representing either an IP name or a DNS entry

ReSpecT Behaviour Specification

$$
\begin{array}{rcl}
\langle \mathit{TCSpecification} \rangle & ::= & \{ \langle \mathit{SpecificationTuple} \rangle \ . \} \\
\langle \mathit{SpecificationTuple} \rangle & ::= & \texttt{reaction(} \\
 & & \quad \langle \mathit{SimpleTCEvent} \rangle \ , \\
 & & \quad [\langle \mathit{Guard} \rangle \ ,] \\
 & & \quad \langle \mathit{Reaction} \rangle \\
 & & \texttt{)}
\end{array}
$$

- a behaviour specification ⟨*TCSpecification*⟩ is a logic theory of FOL tuples `reaction/3`
- a specification tuple contains an event descriptor ⟨*SimpleTCEvent*⟩, a guard ⟨*Guard*⟩ (optional), and a sequence ⟨*Reaction*⟩ of reaction goals
    - a `reaction/2` specification tuple implicitly defines an empty guard

ReSpecT Behaviour Specification

$$
\begin{array}{rcl}
\langle \mathit{TCSpecification} \rangle & ::= & \{ \langle \mathit{SpecificationTuple} \rangle \; . \} \\
\langle \mathit{SpecificationTuple} \rangle & ::= & \texttt{reaction(} \\
 & & \quad \langle \mathit{SimpleTCEvent} \rangle \; , \\
 & & \quad [\langle \mathit{Guard} \rangle \; ,] \\
 & & \quad \langle \mathit{Reaction} \rangle \\
 & & \texttt{)}
\end{array}
$$

- a behaviour specification ⟨*TCSpecification*⟩ is a logic theory of FOL tuples `reaction/3`
- a specification tuple contains an event descriptor ⟨*SimpleTCEvent*⟩, a guard ⟨*Guard*⟩ (optional), and a sequence ⟨*Reaction*⟩ of reaction goals
  - a `reaction/2` specification tuple implicitly defines an empty guard

ReSpecT Behaviour Specification

$$
\begin{array}{rcl}
\langle \mathit{TCSpecification} \rangle & ::= & \{ \langle \mathit{SpecificationTuple} \rangle \; . \} \\
\langle \mathit{SpecificationTuple} \rangle & ::= & \texttt{reaction(} \\
 & & \quad \langle \mathit{SimpleTCEvent} \rangle \; , \\
 & & \quad [\langle \mathit{Guard} \rangle \; ,] \\
 & & \quad \langle \mathit{Reaction} \rangle \\
 & & \texttt{)}
\end{array}
$$

- a behaviour specification ⟨*TCSpecification*⟩ is a logic theory of FOL tuples `reaction/3`
- a specification tuple contains an event descriptor ⟨*SimpleTCEvent*⟩, a guard ⟨*Guard*⟩ (optional), and a sequence ⟨*Reaction*⟩ of reaction goals
  - a `reaction/2` specification tuple implicitly defines an empty guard

ReSpecT Event Descriptor

$$\langle \mathit{SimpleTCEvent} \rangle \quad ::= \quad \langle \mathit{SimpleTCPredicate} \rangle \texttt{(} \langle \mathit{Tuple} \rangle \texttt{)} \mid \texttt{time(} \langle \mathit{Time} \rangle \texttt{)}$$

- an event descriptor ⟨*SimpleTCEvent*⟩ is either the invocation of a primitive ⟨*SimpleTCPredicate*⟩ (⟨*Tuple*⟩) or a time event `time(` ⟨*Time*⟩ `)`
  - more generally, a time event could become the descriptor of an environment-related event
- an event descriptor ⟨*SimpleTCEvent*⟩ is used to match with with *admissible events*

ReSpecT Event Descriptor

$$\langle \mathit{SimpleTCEvent} \rangle \quad ::= \quad \langle \mathit{SimpleTCPredicate} \rangle \, (\, \langle \mathit{Tuple} \rangle \,) \mid \texttt{time}(\, \langle \mathit{Time} \rangle \,)$$

- an event descriptor ⟨*SimpleTCEvent*⟩ is either the invocation of a primitive ⟨*SimpleTCPredicate*⟩ (⟨*Tuple*⟩) or a time event `time`(⟨*Time*⟩)
  - more generally, a time event could become the descriptor of an environment-related event
- an event descriptor ⟨*SimpleTCEvent*⟩ is used to match with with *admissible events*

ReSpecT Event Descriptor

$$\langle \mathit{SimpleTCEvent} \rangle \quad ::= \quad \langle \mathit{SimpleTCPredicate} \rangle \,(\, \langle \mathit{Tuple} \rangle \,)\ |$$
$$\texttt{time(}\, \langle \mathit{Time} \rangle \,\texttt{)}$$

- an event descriptor $\langle \mathit{SimpleTCEvent} \rangle$ is either the invocation of a primitive $\langle \mathit{SimpleTCPredicate} \rangle$ ($\langle \mathit{Tuple} \rangle$) or a time event `time(` $\langle \mathit{Time} \rangle$ `)`
    - more generally, a time event could become the descriptor of an environment-related event
- an event descriptor $\langle \mathit{SimpleTCEvent} \rangle$ is used to match with with *admissible events*

ReSpecT Admissible Event

$$\begin{aligned}
\langle \textit{GeneralTCEvent} \rangle &::= \langle \textit{StartCause} \rangle \texttt{,} \langle \textit{Cause} \rangle \texttt{,} \langle \textit{TCCycleResult} \rangle \\
\langle \textit{StartCause} \rangle , \langle \textit{Cause} \rangle &::= \langle \textit{SimpleTCEvent} \rangle \texttt{,} \langle \textit{Source} \rangle \texttt{,} \langle \textit{Target} \rangle \texttt{,} \langle \textit{Time} \rangle \\
\langle \textit{Source} \rangle , \langle \textit{Target} \rangle &::= \langle \textit{ProcessIdentifier} \rangle \mid \langle \textit{TCIdentifier} \rangle \\
\langle \textit{ProcessIdentifier} \rangle &::= \langle \textit{ProcessName} \rangle \,\texttt{@}\, \langle \textit{NetworkLocation} \rangle \\
\langle \textit{ProcessName} \rangle &\ \text{is}\ \ \text{a Prolog ground term} \\
\langle \textit{TCCycleResult} \rangle &::= \bot \mid \{\langle \textit{Tuple} \rangle\}
\end{aligned}$$

- an admissible event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when a process invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to a process' primitive invocation upon another tuple centre
- a reaction specification tuple `reaction(`E`,`G`,`R`)` and an admissible event ϵ match if E unifies with $\epsilon.\langle \textit{Cause} \rangle.\langle \textit{SimpleTCEvent} \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Admissible Event

$$
\begin{array}{rcl}
\langle \mathit{GeneralTCEvent} \rangle & ::= & \langle \mathit{StartCause} \rangle \texttt{,} \langle \mathit{Cause} \rangle \texttt{,} \langle \mathit{TCCycleResult} \rangle \\
\langle \mathit{StartCause} \rangle , \langle \mathit{Cause} \rangle & ::= & \langle \mathit{SimpleTCEvent} \rangle \texttt{,} \langle \mathit{Source} \rangle \texttt{,} \langle \mathit{Target} \rangle \texttt{,} \langle \mathit{Time} \rangle \\
\langle \mathit{Source} \rangle , \langle \mathit{Target} \rangle & ::= & \langle \mathit{ProcessIdentifier} \rangle \mid \langle \mathit{TCIdentifier} \rangle \\
\langle \mathit{ProcessIdentifier} \rangle & ::= & \langle \mathit{ProcessName} \rangle \texttt{@} \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{ProcessName} \rangle & \text{is} & \text{a Prolog ground term} \\
\langle \mathit{TCCycleResult} \rangle & ::= & \bot \mid \{\langle \mathit{Tuple} \rangle\}
\end{array}
$$

- an admissible event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when a process invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to a process' primitive invocation upon another tuple centre
- a reaction specification tuple `reaction(`E`,`G`,`R`)` and an admissible event ϵ match if E unifies with $\epsilon.\langle \mathit{Cause} \rangle.\langle \mathit{SimpleTCEvent} \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Admissible Event

$$
\begin{array}{rcl}
\langle GeneralTCEvent \rangle & ::= & \langle StartCause \rangle \; \texttt{,} \; \langle Cause \rangle \; \texttt{,} \; \langle TCCycleResult \rangle \\
\langle StartCause \rangle \, , \langle Cause \rangle & ::= & \langle SimpleTCEvent \rangle \; \texttt{,} \; \langle Source \rangle \; \texttt{,} \; \langle Target \rangle \; \texttt{,} \; \langle Time \rangle \\
\langle Source \rangle \, , \langle Target \rangle & ::= & \langle ProcessIdentifier \rangle \mid \langle TCIdentifier \rangle \\
\langle ProcessIdentifier \rangle & ::= & \langle ProcessName \rangle \, \texttt{@} \, \langle NetworkLocation \rangle \\
\langle ProcessName \rangle & \text{is} & \text{a Prolog ground term} \\
\langle TCCycleResult \rangle & ::= & \perp \mid \{\langle Tuple \rangle\}
\end{array}
$$

- an admissible event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when a process invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to a process' primitive invocation upon another tuple centre
- a reaction specification tuple `reaction(`E`,`G`,`R`)` and an admissible event ϵ match if E unifies with $\epsilon.\langle Cause \rangle.\langle SimpleTCEvent \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Admissible Event

$$
\begin{array}{rcl}
\langle \mathit{GeneralTCEvent} \rangle & ::= & \langle \mathit{StartCause} \rangle \texttt{,} \langle \mathit{Cause} \rangle \texttt{,} \langle \mathit{TCCycleResult} \rangle \\
\langle \mathit{StartCause} \rangle , \langle \mathit{Cause} \rangle & ::= & \langle \mathit{SimpleTCEvent} \rangle \texttt{,} \langle \mathit{Source} \rangle \texttt{,} \langle \mathit{Target} \rangle \texttt{,} \langle \mathit{Time} \rangle \\
\langle \mathit{Source} \rangle , \langle \mathit{Target} \rangle & ::= & \langle \mathit{ProcessIdentifier} \rangle \mid \langle \mathit{TCIdentifier} \rangle \\
\langle \mathit{ProcessIdentifier} \rangle & ::= & \langle \mathit{ProcessName} \rangle \texttt{@} \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{ProcessName} \rangle & \text{is} & \text{a Prolog ground term} \\
\langle \mathit{TCCycleResult} \rangle & ::= & \bot \mid \{\langle \mathit{Tuple} \rangle\}
\end{array}
$$

- an admissible event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when a process invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to a process' primitive invocation upon another tuple centre
- a reaction specification tuple `reaction(E,G,R)` and an admissible event ϵ match if E unifies with $\epsilon.\langle \mathit{Cause} \rangle.\langle \mathit{SimpleTCEvent} \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Admissible Event

$$\begin{array}{rcl}
\langle \mathit{GeneralTCEvent} \rangle & ::= & \langle \mathit{StartCause} \rangle \texttt{,} \langle \mathit{Cause} \rangle \texttt{,} \langle \mathit{TCCycleResult} \rangle \\
\langle \mathit{StartCause} \rangle , \langle \mathit{Cause} \rangle & ::= & \langle \mathit{SimpleTCEvent} \rangle \texttt{,} \langle \mathit{Source} \rangle \texttt{,} \langle \mathit{Target} \rangle \texttt{,} \langle \mathit{Time} \rangle \\
\langle \mathit{Source} \rangle , \langle \mathit{Target} \rangle & ::= & \langle \mathit{ProcessIdentifier} \rangle \mid \langle \mathit{TCIdentifier} \rangle \\
\langle \mathit{ProcessIdentifier} \rangle & ::= & \langle \mathit{ProcessName} \rangle \texttt{@} \langle \mathit{NetworkLocation} \rangle \\
\langle \mathit{ProcessName} \rangle & \text{is} & \text{a Prolog ground term} \\
\langle \mathit{TCCycleResult} \rangle & ::= & \bot \mid \{\langle \mathit{Tuple} \rangle\}
\end{array}$$

- an admissible event descriptor includes its prime cause, its immediate cause, and the result of the tuple centre response
  - prime cause and immediate cause may coincide—such as when a process invocation reaches its target tuple centre
  - or, they might be different—such as when a link primitive is invoked by a tuple centre reacting to a process' primitive invocation upon another tuple centre
- a reaction specification tuple `reaction(E,G,R)` and an admissible event ϵ match if E unifies with $\epsilon.\langle \mathit{Cause} \rangle.\langle \mathit{SimpleTCEvent} \rangle$
- the result is undefined in the invocation stage, whereas it is defined in the completion stage

ReSpecT Guards

$$\langle \mathit{Guard} \rangle ::= \langle \mathit{GuardPredicate} \rangle \mid (\, \langle \mathit{GuardPredicate} \rangle \,\{ , \langle \mathit{GuardPredicate} \rangle \}\,)$$

$$\langle \mathit{GuardPredicate} \rangle ::= \texttt{request} \mid \texttt{response} \mid \texttt{success} \mid \texttt{failure} \mid \texttt{endo} \mid \texttt{exo} \mid \texttt{intra} \mid \texttt{inter} \mid \texttt{from\_agent} \mid \texttt{to\_agent} \mid \texttt{from\_tc} \mid \texttt{to\_tc} \mid \texttt{before(}\,\langle \mathit{Time} \rangle\,\texttt{)} \mid \texttt{after(}\,\langle \mathit{Time} \rangle\,\texttt{)}$$

$\langle \mathit{Time} \rangle$ is a non-negative integer

- A triggered reaction is actually executed only if its guard is true
- All guard predicates are ground ones, so their have always a success / failure semantics
- Guard predicates concern properties of the event, so they can be used to further select some classes of events after the initial matching between the admissible event and the event descriptor

ReSpecT Guards

$\langle \mathit{Guard} \rangle$ ::= $\langle \mathit{GuardPredicate} \rangle$ |
($\langle \mathit{GuardPredicate} \rangle$ { , $\langle \mathit{GuardPredicate} \rangle$ })

$\langle \mathit{GuardPredicate} \rangle$ ::= `request` | `response` | `success` | `failure` |
`endo` | `exo` | `intra` | `inter` |
`from_agent` | `to_agent` | `from_tc` | `to_tc` |
`before(` $\langle \mathit{Time} \rangle$ `)` | `after(` $\langle \mathit{Time} \rangle$ `)`

$\langle \mathit{Time} \rangle$ is a non-negative integer

- A triggered reaction is actually executed only if its guard is true
- All guard predicates are ground ones, so their have always a success / failure semantics
- Guard predicates concern properties of the event, so they can be used to further select some classes of events after the initial matching between the admissible event and the event descriptor

ReSpecT Guards

$\langle Guard \rangle$ ::= $\langle GuardPredicate \rangle$ |
($\langle GuardPredicate \rangle$ { , $\langle GuardPredicate \rangle$ })

$\langle GuardPredicate \rangle$ ::= `request` | `response` | `success` | `failure` |
`endo` | `exo` | `intra` | `inter` |
`from_agent` | `to_agent` | `from_tc` | `to_tc` |
`before(` $\langle Time \rangle$ `)` | `after(` $\langle Time \rangle$ `)`

$\langle Time \rangle$ is a non-negative integer

- A triggered reaction is actually executed only if its guard is true
- All guard predicates are ground ones, so their have always a success / failure semantics
- Guard predicates concern properties of the event, so they can be used to further select some classes of events after the initial matching between the admissible event and the event descriptor

Semantics of Guard Predicates in ReSpecT

Guard atom	True if
$Guard(\epsilon, (g, G))$	$Guard(\epsilon, g) \wedge Guard(\epsilon, G)$
$Guard(\epsilon, \texttt{endo})$	$\epsilon.Cause.Source = c$
$Guard(\epsilon, \texttt{exo})$	$\epsilon.Cause.Source \neq c$
$Guard(\epsilon, \texttt{intra})$	$\epsilon.Cause.Target = c$
$Guard(\epsilon, \texttt{inter})$	$\epsilon.Cause.Target \neq c$
$Guard(\epsilon, \texttt{from\_agent})$	$\epsilon.Cause.Source$ *is an agent*
$Guard(\epsilon, \texttt{to\_agent})$	$\epsilon.Cause.Target$ *is an agent*
$Guard(\epsilon, \texttt{from\_tc})$	$\epsilon.Cause.Source$ *is a tuple centre*
$Guard(\epsilon, \texttt{to\_tc})$	$\epsilon.Cause.Target$ *is a tuple centre*
$Guard(\epsilon, \texttt{before}(t))$	$\epsilon.Cause.Time < t$
$Guard(\epsilon, \texttt{after}(t))$	$\epsilon.Cause.Time > t$
$Guard(\epsilon, \texttt{request})$	$\epsilon.TCCycleResult$ *is undefined*
$Guard(\epsilon, \texttt{response})$	$\epsilon.TCCycleResult$ *is defined*
$Guard(\epsilon, \texttt{success})$	$\epsilon.TCCycleResult \neq \perp$
$Guard(\epsilon, \texttt{failure})$	$\epsilon.TCCycleResult = \perp$

⟨*GuardPredicate*⟩ aliases

`request` `invocation`, `inv`, `req`, `pre`

`response` `completion`, `compl`, `resp`, `post`

`before(`*Time*`),after(`*Time'*`)` `between(`*Time*`,`*Time'*`)`

`from_agent,to_tc` `operation`

`from_tc,to_tc,endo,inter` `link_out`

`from_tc,to_tc,exo,intra` `link_in`

`from_tc,to_tc,endo,intra` `internal`

⟨*GuardPredicate*⟩ aliases

`request`	`invocation`, `inv`, `req`, `pre`
`response`	`completion`, `compl`, `resp`, `post`
`before(`*Time*`),after(`*Time'*`)`	`between(`*Time*`,`*Time'*`)`
`from_agent,to_tc`	`operation`
`from_tc,to_tc,endo,inter`	`link_out`
`from_tc,to_tc,exo,intra`	`link_in`
`from_tc,to_tc,endo,intra`	`internal`

⟨*GuardPredicate*⟩ aliases

`request`	`invocation`, `inv`, `req`, `pre`
`response`	`completion`, `compl`, `resp`, `post`
`before(`*Time*`),after(`*Time'*`)`	`between(`*Time*`,`*Time'*`)`
`from_agent,to_tc`	`operation`
`from_tc,to_tc,endo,inter`	`link_out`
`from_tc,to_tc,exo,intra`	`link_in`
`from_tc,to_tc,endo,intra`	`internal`

⟨*GuardPredicate*⟩ aliases

`request`	`invocation`, `inv`, `req`, `pre`
`response`	`completion`, `compl`, `resp`, `post`
`before(`*Time*`),after(`*Time'*`)`	`between(`*Time*`,`*Time'*`)`
`from_agent,to_tc`	`operation`
`from_tc,to_tc,endo,inter`	`link_out`
`from_tc,to_tc,exo,intra`	`link_in`
`from_tc,to_tc,endo,intra`	`internal`

⟨*GuardPredicate*⟩ aliases

`request` invocation, inv, req, pre

`response` completion, compl, resp, post

`before(`*Time*`),after(`*Time'*`)` between(*Time*,*Time'*)

`from_agent,to_tc` operation

`from_tc,to_tc,endo,inter` link_out

`from_tc,to_tc,exo,intra` link_in

`from_tc,to_tc,endo,intra` internal

⟨*GuardPredicate*⟩ aliases

`request`	`invocation`, `inv`, `req`, `pre`
`response`	`completion`, `compl`, `resp`, `post`
`before(`*Time*`),after(`*Time'*`)`	`between(`*Time*`,`*Time'*`)`
`from_agent,to_tc`	`operation`
`from_tc,to_tc,endo,inter`	`link_out`
`from_tc,to_tc,exo,intra`	`link_in`
`from_tc,to_tc,endo,intra`	`internal`

⟨*GuardPredicate*⟩ aliases

`request`	`invocation`, `inv`, `req`, `pre`
`response`	`completion`, `compl`, `resp`, `post`
`before(`*Time*`),after(`*Time'*`)`	`between(`*Time*`,`*Time'*`)`
`from_agent,to_tc`	`operation`
`from_tc,to_tc,endo,inter`	`link_out`
`from_tc,to_tc,exo,intra`	`link_in`
`from_tc,to_tc,endo,intra`	`internal`

ReSpecT Reactions

$$\langle \mathit{Reaction} \rangle ::= \langle \mathit{ReactionGoal} \rangle \mid (\, \langle \mathit{ReactionGoal} \rangle \, \{ \, , \langle \mathit{ReactionGoal} \rangle \} \,)$$

$$\langle \mathit{ReactionGoal} \rangle ::= \langle \mathit{TCPredicate} \rangle \, (\, \langle \mathit{Tuple} \rangle \,) \mid \langle \mathit{ObservationPredicate} \rangle \, (\, \langle \mathit{Tuple} \rangle \,) \mid \langle \mathit{Computation} \rangle \mid (\, \langle \mathit{ReactionGoal} \rangle \, ; \, \langle \mathit{ReactionGoal} \rangle \,)$$

$$\langle \mathit{TCPredicate} \rangle ::= \langle \mathit{SimpleTCPredicate} \rangle \mid \langle \mathit{TCLinkPredicate} \rangle$$

$$\langle \mathit{TCLinkPredicate} \rangle ::= \langle \mathit{TCIdentifier} \rangle \, ? \, \langle \mathit{SimpleTCPredicate} \rangle$$

- A reaction goal is either a primitive invocation (possibly, a link), 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

ReSpecT Reactions

$$
\begin{array}{rcl}
\langle \mathit{Reaction} \rangle & ::= & \langle \mathit{ReactionGoal} \rangle \mid \\
 & & (\ \langle \mathit{ReactionGoal} \rangle \ \{\ ,\ \langle \mathit{ReactionGoal} \rangle \} \ ) \\
\langle \mathit{ReactionGoal} \rangle & ::= & \langle \mathit{TCPredicate} \rangle \ (\ \langle \mathit{Tuple} \rangle \ ) \mid \\
 & & \langle \mathit{ObservationPredicate} \rangle \ (\ \langle \mathit{Tuple} \rangle \ ) \mid \\
 & & \langle \mathit{Computation} \rangle \mid \\
 & & (\ \langle \mathit{ReactionGoal} \rangle \ ;\ \langle \mathit{ReactionGoal} \rangle \ ) \\
\langle \mathit{TCPredicate} \rangle & ::= & \langle \mathit{SimpleTCPredicate} \rangle \mid \langle \mathit{TCLinkPredicate} \rangle \\
\langle \mathit{TCLinkPredicate} \rangle & ::= & \langle \mathit{TCIdentifier} \rangle \ ?\ \langle \mathit{SimpleTCPredicate} \rangle
\end{array}
$$

- A reaction goal is either a primitive invocation (possibly, a link), 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

ReSpecT Tuple Centre Predicates

$\langle SimpleTCPredicate \rangle$ ::= $\langle TCStatePredicate \rangle$ | $\langle TCForgePredicate \rangle$

$\langle TCStatePredicate \rangle$ ::= `in` | `inp` | `rd` | `rdp` | `out` | `no` | `get` | `set`

$\langle TCForgePredicate \rangle$ ::= $\langle TCStatePredicate \rangle$`_s`

- Tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- The same predicates are substantially used for changing the specification state, with essentially the same semantics
  - *pred_s* invocations affect the specification state, and can be used within reactions, also as links
- `no` works as a test for absence, `get` and `set` work on the overall theory (either the one of ordinary tuples, or the one of specification tuples)

ReSpecT Tuple Centre Predicates

$\langle SimpleTCPredicate \rangle$ ::= $\langle TCStatePredicate \rangle$ | $\langle TCForgePredicate \rangle$

$\langle TCStatePredicate \rangle$ ::= `in` | `inp` | `rd` | `rdp` | `out` | `no` | `get` | `set`

$\langle TCForgePredicate \rangle$ ::= $\langle TCStatePredicate \rangle$`_s`

- Tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- The same predicates are substantially used for changing the specification state, with essentially the same semantics
  - *pred*_s invocations affect the specification state, and can be used within reactions, also as links
- `no` works as a test for absence, `get` and `set` work on the overall theory (either the one of ordinary tuples, or the one of specification tuples)

ReSpecT Tuple Centre Predicates

$\langle$*SimpleTCPredicate*$\rangle$ ::= $\langle$*TCStatePredicate*$\rangle$ | $\langle$*TCForgePredicate*$\rangle$

$\langle$*TCStatePredicate*$\rangle$ ::= `in` | `inp` | `rd` | `rdp` | `out` | `no` | `get` | `set`

$\langle$*TCForgePredicate*$\rangle$ ::= $\langle$*TCStatePredicate*$\rangle$`_s`

- Tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- The same predicates are substantially used for changing the specification state, with essentially the same semantics
  - *pred*_s invocations affect the specification state, and can be used within reactions, also as links
- no works as a test for absence, get and set work on the overall theory (either the one of ordinary tuples, or the one of specification tuples)

ReSpecT Tuple Centre Predicates

$\langle \textit{SimpleTCPredicate} \rangle$ ::= $\langle \textit{TCStatePredicate} \rangle$ | $\langle \textit{TCForgePredicate} \rangle$

$\langle \textit{TCStatePredicate} \rangle$ ::= `in` | `inp` | `rd` | `rdp` | `out` | `no` | `get` | `set`

$\langle \textit{TCForgePredicate} \rangle$ ::= $\langle \textit{TCStatePredicate} \rangle$`_s`

- Tuple centre predicates are uniformly used for agent invocations, internal operations, and link invocations
- The same predicates are substantially used for changing the specification state, with essentially the same semantics
  - *pred*`_s` invocations affect the specification state, and can be used within reactions, also as links
- `no` works as a test for absence, `get` and `set` work on the overall theory (either the one of ordinary tuples, or the one of specification tuples)

ReSpecT Observation Predicates

⟨*ObservationPredicate*⟩	::=	⟨*EventView*⟩_⟨*EventInformation*⟩
⟨*EventView*⟩	::=	`current` | `event` | `start`
⟨*EventInformation*⟩	::=	`predicate` | `tuple` | `source` | `target` | `time`

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

  `predicate` `pred`, `call`; *deprecated*: `operation`, `op`

  `tuple` `arg`

  `source` `from`

  `target` `to`

ReSpecT Observation Predicates

$$\langle \mathit{ObservationPredicate} \rangle ::= \langle \mathit{EventView} \rangle\_\langle \mathit{EventInformation} \rangle$$
$$\langle \mathit{EventView} \rangle ::= \texttt{current} \mid \texttt{event} \mid \texttt{start}$$
$$\langle \mathit{EventInformation} \rangle ::= \texttt{predicate} \mid \texttt{tuple} \mid \texttt{source} \mid \texttt{target} \mid \texttt{time}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

  `predicate` `pred`, `call`; *deprecated*: `operation`, `op`
  `tuple` `arg`
  `source` `from`
  `target` `to`

ReSpecT Observation Predicates

$$\langle \mathit{ObservationPredicate} \rangle \ ::= \ \langle \mathit{EventView} \rangle\_\langle \mathit{EventInformation} \rangle$$
$$\langle \mathit{EventView} \rangle \ ::= \ \texttt{current} \mid \texttt{event} \mid \texttt{start}$$
$$\langle \mathit{EventInformation} \rangle \ ::= \ \texttt{predicate} \mid \texttt{tuple} \mid \texttt{source} \mid \texttt{target} \mid \texttt{time}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

  `predicate` `pred`, `call`; *deprecated*: `operation`, `op`

  `tuple` `arg`

  `source` `from`

  `target` `to`

ReSpecT Observation Predicates

$$\langle \mathit{ObservationPredicate} \rangle ::= \langle \mathit{EventView} \rangle\_\langle \mathit{EventInformation} \rangle$$

$$\langle \mathit{EventView} \rangle ::= \texttt{current} \mid \texttt{event} \mid \texttt{start}$$

$$\langle \mathit{EventInformation} \rangle ::= \texttt{predicate} \mid \texttt{tuple} \mid \texttt{source} \mid \texttt{target} \mid \texttt{time}$$

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

  `predicate` `pred`, `call`; *deprecated*: `operation`, `op`

  `tuple` `arg`

  `source` `from`

  `target` `to`

ReSpecT Observation Predicates

⟨*ObservationPredicate*⟩	::=	⟨*EventView*⟩_⟨*EventInformation*⟩
⟨*EventView*⟩	::=	`current` | `event` | `start`
⟨*EventInformation*⟩	::=	`predicate` | `tuple` | `source` | `target` | `time`

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- ⟨*EventInformation*⟩ aliases

  `predicate` `pred`, `call`; *deprecated*: `operation`, `op`
  `tuple` `arg`
  `source` `from`
  `target` `to`

ReSpecT Observation Predicates

$\langle ObservationPredicate \rangle$ ::= $\langle EventView \rangle$_$\langle EventInformation \rangle$

$\langle EventView \rangle$ ::= `current` | `event` | `start`

$\langle EventInformation \rangle$ ::= `predicate` | `tuple` | `source` | `target` | `time`

- `event` & `start` clearly refer to immediate and prime cause, respectively—`current` refers to what is currently happening, whenever this means something useful
- $\langle EventInformation \rangle$ aliases

  `predicate` `pred`, `call`; *deprecated*: `operation`, `op`
  `tuple` `arg`
  `source` `from`
  `target` `to`

Semantics of Observation Predicates

$$\langle (r, R), Tu, \Sigma, Re, Out \rangle_\epsilon \longrightarrow_e \langle R\theta, Tu, \Sigma, Re, Out \rangle_\epsilon$$

r	where
`event_predicate(Obs)`	$\theta = mgu(\epsilon.Cause.SimpleTCEvent.SimpleTCPredicate, \texttt{Obs})$
`event_tuple(Obs)`	$\theta = mgu(\epsilon.Cause.SimpleTCEvent.Tuple, \texttt{Obs})$
`event_source(Obs)`	$\theta = mgu(\epsilon.Cause.Source, \texttt{Obs})$
`event_target(Obs)`	$\theta = mgu(\epsilon.Cause.Target, \texttt{Obs})$
`event_time(Obs)`	$\theta = mgu(\epsilon.Cause.Time, \texttt{Obs})$
`start_predicate(Obs)`	$\theta = mgu(\epsilon.StartCause.SimpleTCEvent.SimpleTCPredicate, \texttt{Obs})$
`start_tuple(Obs)`	$\theta = mgu(\epsilon.StartCause.SimpleTCEvent.Tuple, \texttt{Obs})$
`start_source(Obs)`	$\theta = mgu(\epsilon.StartCause.Source, \texttt{Obs})$
`start_target(Obs)`	$\theta = mgu(\epsilon.StartCause.Target, \texttt{Obs})$
`start_time(Obs)`	$\theta = mgu(\epsilon.StartCause.Time, \texttt{Obs})$
`current_predicate(Obs)`	$\theta = mgu(\texttt{current\_predicate}, \texttt{Obs})$
`current_tuple(Obs)`	$\theta = mgu(\texttt{Obs}, \texttt{Obs}) = \{\}$
`current_source(Obs)`	$\theta = mgu(c, \texttt{Obs})$
`current_target(Obs)`	$\theta = mgu(c, \texttt{Obs})$
`current_time(Obs)`	$\theta = mgu(nc, \texttt{Obs})$

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Properties of ReSpecT Tuple Centres

- ReSpecT tuple centres
  - encapsulate knowledge in terms of logic tuples
  - encapsulates behaviour in terms of ReSpecT specifications
- ReSpecT tuple centres are
  - inspectable
  - malleable
  - linkable
  - situated
    - time
    - external resources

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
  - for knowledge, information, messages, communication
  - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
  - for behaviour, function, coordination
  - working as the (logic) *theory of coordination* for distributed systems

- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space, `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
  - for knowledge, information, messages, communication
  - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
  - for behaviour, function, coordination
  - working as the (logic) *theory of coordination* for distributed systems

- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
  - for knowledge, information, messages, communication
  - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
  - for behaviour, function, coordination
  - working as the (logic) *theory of coordination* for distributed systems

- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples
  - **tuple space** ordinary (logic) tuples
    - for knowledge, information, messages, communication
    - working as the (logic) *theory of communication* for distributed systems
  - **specification space** specification (logic, ReSpecT) tuples
    - for behaviour, function, coordination
    - working as the (logic) *theory of coordination* for distributed systems
- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
  - for knowledge, information, messages, communication
  - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
  - for behaviour, function, coordination
  - working as the (logic) *theory of coordination* for distributed systems

- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space, `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
  - for knowledge, information, messages, communication
  - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
  - for behaviour, function, coordination
  - working as the (logic) *theory of coordination* for distributed systems

- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples
  - **tuple space** ordinary (logic) tuples
    - for knowledge, information, messages, communication
    - working as the (logic) *theory of communication* for distributed systems
  - **specification space** specification (logic, ReSpecT) tuples
    - for behaviour, function, coordination
    - working as the (logic) *theory of coordination* for distributed systems
- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space, `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
    - for knowledge, information, messages, communication
    - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
    - for behaviour, function, coordination
    - working as the (logic) *theory of coordination* for distributed systems

- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
    - for knowledge, information, messages, communication
    - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
    - for behaviour, function, coordination
    - working as the (logic) *theory of coordination* for distributed systems

- Both spaces are inspectable
    - by engineers, via ReSpecT inspectors
    - by processes, via `rd` & `no` primitives
        - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
        - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
  - for knowledge, information, messages, communication
  - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
  - for behaviour, function, coordination
  - working as the (logic) *theory of coordination* for distributed systems

- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
    - for knowledge, information, messages, communication
    - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
    - for behaviour, function, coordination
    - working as the (logic) *theory of coordination* for distributed systems
- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Inspectability of ReSpecT Tuple Centres

- ReSpecT tuple centres: twofold space for tuples

  tuple space ordinary (logic) tuples
  - for knowledge, information, messages, communication
  - working as the (logic) *theory of communication* for distributed systems

  specification space specification (logic, ReSpecT) tuples
  - for behaviour, function, coordination
  - working as the (logic) *theory of coordination* for distributed systems

- Both spaces are inspectable
  - by engineers, via ReSpecT inspectors
  - by processes, via `rd` & `no` primitives
    - `rd` & `no` for the tuple space; `rd_s` & `no_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Malleability of ReSpecT Tuple Centres

- The behaviour of a ReSpecT tuple centre is defined by the ReSpecT tuples in the specification space
  - it can be adapted / changed by changing its ReSpecT specification
- ReSpecT tuple centres are malleable
  - by engineers, via ReSpecT tools
  - by processes, via `in` & `out` primitives
    - `in` & `out` for the tuple space; `in_s` & `out_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Malleability of ReSpecT Tuple Centres

- The behaviour of a ReSpecT tuple centre is defined by the ReSpecT tuples in the specification space
  - it can be adapted / changed by changing its ReSpecT specification
- ReSpecT tuple centres are malleable
  - by engineers, via ReSpecT tools
  - by processes, via `in` & `out` primitives
    - `in` & `out` for the tuple space, `in_s` & `out_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Malleability of ReSpecT Tuple Centres

- The behaviour of a ReSpecT tuple centre is defined by the ReSpecT tuples in the specification space
  - it can be adapted / changed by changing its ReSpecT specification
- ReSpecT tuple centres are malleable
  - by engineers, via ReSpecT tools
  - by processes, via `in` & `out` primitives
    - `in` & `out` for the tuple space; `in_s` & `out_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Malleability of ReSpecT Tuple Centres

- The behaviour of a ReSpecT tuple centre is defined by the ReSpecT tuples in the specification space
  - it can be adapted / changed by changing its ReSpecT specification
- ReSpecT tuple centres are malleable
  - by engineers, via ReSpecT tools
  - by processes, via `in` & `out` primitives
    - `in` & `out` for the tuple space; `in_s` & `out_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Malleability of ReSpecT Tuple Centres

- The behaviour of a ReSpecT tuple centre is defined by the ReSpecT tuples in the specification space
  - it can be adapted / changed by changing its ReSpecT specification
- ReSpecT tuple centres are malleable
  - by engineers, via ReSpecT tools
  - by processes, via `in` & `out` primitives
    - `in` & `out` for the tuple space; `in_s` & `out_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Malleability of ReSpecT Tuple Centres

- The behaviour of a ReSpecT tuple centre is defined by the ReSpecT tuples in the specification space
  - it can be adapted / changed by changing its ReSpecT specification
- ReSpecT tuple centres are malleable
  - by engineers, via ReSpecT tools
  - by processes, via `in` & `out` primitives
    - `in` & `out` for the tuple space; `in_s` & `out_s` for the specification space
    - either directly or indirectly, through either a coordination primitive, or another tuple centre

Malleability of ReSpecT Tuple Centres

- The behaviour of a ReSpecT tuple centre is defined by the ReSpecT tuples in the specification space
    - it can be adapted / changed by changing its ReSpecT specification
- ReSpecT tuple centres are malleable
    - by engineers, via ReSpecT tools
    - by processes, via `in` & `out` primitives
        - `in` & `out` for the tuple space; `in_s` & `out_s` for the specification space
        - either directly or indirectly, through either a coordination primitive, or another tuple centre

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including out / out_s
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
    $< \mathit{TCIdentifier} > @ < \mathit{NetworkLocation} > ? < \mathit{SimpleTCPredicate} >$
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including out / out_s
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
    < *TCIdentifier* > @ < *NetworkLocation* >? < *SimpleTCPredicate* >
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including out / out_s
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
    < *TCIdentifier* > @ < *NetworkLocation* >? < *SimpleTCPredicate* >
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
    $< \mathit{TCIdentifier} > @ < \mathit{NetworkLocation} > ? < \mathit{SimpleTCPredicate} >$
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
    < *TCIdentifier* > @ < *NetworkLocation* > ? < *SimpleTCPredicate* >
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions

    < *TCIdentifier* > @ < *NetworkLocation* >? < *SimpleTCPredicate* >
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
    < *TCIdentifier* > @ < *NetworkLocation* > ? < *SimpleTCPredicate* >
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
    < *TCIdentifier* > @ < *NetworkLocation* > ? < *SimpleTCPredicate* >
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions

    < *TCIdentifier* > @ < *NetworkLocation* > ? < *SimpleTCPredicate* >
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions

    < *TCIdentifier* > @ < *NetworkLocation* >? < *SimpleTCPredicate* >
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
  - all primitives are asynchronous
    - so they do not affect the transactional semantics of reactions
  - all primitives have a request / response semantics
    - including `out` / `out_s`
    - so reactions can be defined to handle both primitive invocations & completions
  - all primitives could be executed within a ReSpecT reaction
    - as either a reaction goal executed within the same tuple centre
    - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
  - by using tuple centre identifiers within ReSpecT reactions
    $< \mathit{TCIdentifier} >$ @ $< \mathit{NetworkLocation} >$? $< \mathit{SimpleTCPredicate} >$
  - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Linkability of ReSpecT Tuple Centres

- Every tuple centre coordination primitive is also an ReSpecT primitive for reaction goals, and a primitive for linking, too
    - all primitives are asynchronous
        - so they do not affect the transactional semantics of reactions
    - all primitives have a request / response semantics
        - including `out` / `out_s`
        - so reactions can be defined to handle both primitive invocations & completions
    - all primitives could be executed within a ReSpecT reaction
        - as either a reaction goal executed within the same tuple centre
        - or as a link primitive invoked upon another tuple centre
- ReSpecT tuple centres are linkable
    - by using tuple centre identifiers within ReSpecT reactions
      $< \mathit{TCIdentifier} >$ `@` $< \mathit{NetworkLocation} >$? $< \mathit{SimpleTCPredicate} >$
    - any ReSpecT reaction can invoke any coordination primitive upon any tuple centre in the network

Outline

Outline

Dining Philosophers in ReSpecT: Starvation?

What is the problem?

- The problem is *time*: no one keeps track of time here, and starvation is a matter of time
- How can we handle time here? Is synchronisation not enough for the purpose?
- Of course not: to avoid problems like starvation, we need the ability of defining *time-dependent* coordination policies

What is the solution?

- In order to define time-dependent coordination policies, a time-aware coordination medium is needed

Dining Philosophers in ReSpecT: Starvation?

What is the problem?

- The problem is *time*: no one keeps track of time here, and starvation is a matter of time
- How can we handle time here? Is synchronisation not enough for the purpose?
- Of course not: to avoid problems like starvation, we need the ability of defining *time-dependent* coordination policies

What is the solution?

- In order to define time-dependent coordination policies, a time-aware coordination medium is needed

Dining Philosophers in ReSpecT: Starvation?

What is the problem?

- The problem is *time*: no one keeps track of time here, and starvation is a matter of time
- How can we handle time here? Is synchronisation not enough for the purpose?
- Of course not: to avoid problems like starvation, we need the ability of defining *time-dependent* coordination policies

What is the solution?

- In order to define time-dependent coordination policies, a time-aware coordination medium is needed

Dining Philosophers in ReSpecT: Starvation?

What is the problem?

- The problem is *time*: no one keeps track of time here, and starvation is a matter of time
- How can we handle time here? Is synchronisation not enough for the purpose?
- Of course not: to avoid problems like starvation, we need the ability of defining *time-dependent* coordination policies

What is the solution?

- In order to define time-dependent coordination policies, a time-aware coordination medium is needed

Dining Philosophers in ReSpecT: Starvation?

What is the problem?

- The problem is *time*: no one keeps track of time here, and starvation is a matter of time
- How can we handle time here? Is synchronisation not enough for the purpose?
- Of course not: to avoid problems like starvation, we need the ability of defining *time-dependent* coordination policies

What is the solution?

- In order to define time-dependent coordination policies, a time-aware coordination medium is needed

Dining Philosophers in ReSpecT: Starvation?

What is the problem?

- The problem is *time*: no one keeps track of time here, and starvation is a matter of time
- How can we handle time here? Is synchronisation not enough for the purpose?
- Of course not: to avoid problems like starvation, we need the ability of defining *time-dependent* coordination policies

What is the solution?

- In order to define time-dependent coordination policies, a time-aware coordination medium is needed

Time-dependent Coordination I

Time-aware coordination media [Omicini et al., 2007]

A time-aware coordination medium for time-dependent coordination policies essentially means

- Time has to be an integral part of the ontology of a coordination medium
- A coordination medium should allow coordination policies to talk about time
- (Physical) time has to be explicitly embedded into the coordination medium working cycle
- A coordination medium should be able to capture time events, and to react appropriately
- A coordination medium should allow coordination policies to be changed over time

Time-dependent Coordination II

Timed ReSpecT [Omicini et al., 2005]

Accordingly, ReSpecT is extended with time

- by introducing some temporal predicates to get information about both tuple-centre and event time
  - `current_time(?Time)`
  - `event_time(?Time)`
  - `before(@Time)`, `after(@Time)`, `between(@MinTime,@MaxTime)`
- by making it possible to specify reactions to the occurrence of *time events*
  - `reaction(time(@Time), Guard, Body).`
- by exploiting malleabilty to allow coordination policies to be changed over time

Timed Dining Philosophers

- An example of time-dependent coordination
- `table` tuple centre stores the maximum amount of time for any process (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by processes through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005]

Timed Dining Philosophers

- An example of time-dependent coordination
- `table` tuple centre stores the maximum amount of time for any process (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the `table` tuple centre
  - late releases (by processes through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005]

Timed Dining Philosophers

- An example of time-dependent coordination
- `table` tuple centre stores the maximum amount of time for any process (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by processes through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005]

Timed Dining Philosophers

- An example of time-dependent coordination
- `table` tuple centre stores the maximum amount of time for any process (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by processes through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005]

Timed Dining Philosophers

- An example of time-dependent coordination
- `table` tuple centre stores the maximum amount of time for any process (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by processes through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005]

Timed Dining Philosophers

- An example of time-dependent coordination
- `table` tuple centre stores the maximum amount of time for any process (philosopher) to use the resource (to eat using chops)
    - in terms of a tuple `max_eating_time(@Time)`
    - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
    - late releases (by processes through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
    - see [Omicini et al., 2005]

Timed Dining Philosophers

- An example of time-dependent coordination
- `table` tuple centre stores the maximum amount of time for any process (philosopher) to use the resource (to eat using chops)
  - in terms of a tuple `max_eating_time(@Time)`
  - if this time expires the locks are automatically released—chopsticks are re-inserted by the table tuple centre
  - late releases (by processes through `seat` tuple centres) are to be ignored—linkability used to make `seat` tuple centres consistent
- With a very simple extension using timed reactions, Distributed Timed Dining Philosophers are done
  - see [Omicini et al., 2005]

Timed Dining Philosophers: Philosopher

```
philosopher(I,J) :-
    think,                          % thinking
    table ? in(chops(I,J)),         % waiting to eat
    eat,                            % eating
    table ? out(chops(I,J)),        % waiting to think
!,  philosopher(I,J).
```

With respect to Dining Philosopher's protocol. . .

. . . this is left unchanged

Timed Dining Philosophers: Philosopher

```
philosopher(I,J) :-
    think,                        % thinking
    table ? in(chops(I,J)),       % waiting to eat
    eat,                          % eating
    table ? out(chops(I,J)),      % waiting to think
!,  philosopher(I,J).
```

With respect to Dining Philosopher's protocol. . .

. . . this is left unchanged

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
   in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (      % (2)
   out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (      % (3)
   in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                 % (4)
   in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                         % (5)
   rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2))
reaction( out(chop(C)), internal, (                         % (5')
   rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C))
reaction( in(chops(C1,C2)), (operation, completion), (      % (6)
   current_time(T), rd(max eating time(Max)), T1 is T + Max,
   out(used(C1,C2,T)),
   out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
   in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
   out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
   in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
   in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
   rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2))
reaction( out(chop(C)), internal, (                          % (5')
   rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C))
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
   current_time(T), rd(max eating time(Max)), T1 is T + Max,
   out(used(C1,C2,T)),
   out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2))
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C))
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (        % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (         % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (         % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                     % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                             % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2))
reaction( out(chop(C)), internal, (                             % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C))
reaction( in(chops(C1,C2)), (operation, completion), (         % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2))
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C))
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2))
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C))
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)) )).
reaction( out(chops(C1,C2)), (operation, completion), (      % (1')
    out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2))) )
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (      % (1)
    in(chops(C1,C2)) )).
reaction( out(chops(C1,C2)), (operation, completion), (      % (1')
    in(used(C1,C2,_)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (       % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                  % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                          % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                          % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (       % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2))) )
```

Timed Dining Philosophers: table ReSpecT Code

```
reaction( out(chops(C1,C2)), (operation, completion), (        % (1)
    in(chops(C1,C2)) )).
reaction( out(chops(C1,C2)), (operation, completion), (        % (1')
    in(used(C1,C2,_)), out(chop(C1)), out(chop(C2)) )).
reaction( in(chops(C1,C2)), (operation, invocation), (         % (2)
    out(required(C1,C2)) )).
reaction( in(chops(C1,C2)), (operation, completion), (         % (3)
    in(required(C1,C2)) )).
reaction( out(required(C1,C2)), internal, (                    % (4)
    in(chop(C1)), in(chop(C2)), out(chops(C1,C2)) )).
reaction( out(chop(C)), internal, (                            % (5)
    rd(required(C,C2)), in(chop(C)), in(chop(C2)), out(chops(C,C2)) )).
reaction( out(chop(C)), internal, (                            % (5')
    rd(required(C1,C)), in(chop(C1)), in(chop(C)), out(chops(C1,C)) )).
reaction( in(chops(C1,C2)), (operation, completion), (         % (6)
    current_time(T), rd(max eating time(Max)), T1 is T + Max,
    out(used(C1,C2,T)),
    out_s(time(T1),(in(used(C1,C2,T)), out(chop(C1)), out(chop(C2)))) )).
```

Timed Dining Philosophers in ReSpecT: Results

Results

- **protocol** no deadlock
- **protocol** fairness
- **protocol** trivial philosopher's interaction protocol
- **tuple centre** shared resources handled properly
- **tuple centre** no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
tuple centre	no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

protocol	no deadlock
protocol	fairness
protocol	trivial philosopher's interaction protocol
tuple centre	shared resources handled properly
tuple centre	no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

- **protocol** no deadlock
- **protocol** fairness
- **protocol** trivial philosopher's interaction protocol
- **tuple centre** shared resources handled properly
- **tuple centre** no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

- **protocol** no deadlock
- **protocol** fairness
- **protocol** trivial philosopher's interaction protocol
- **tuple centre** shared resources handled properly
- **tuple centre** no starvation

Timed Dining Philosophers in ReSpecT: Results

Results

- **protocol** no deadlock
- **protocol** fairness
- **protocol** trivial philosopher's interaction protocol
- **tuple centre** shared resources handled properly
- **tuple centre** no starvation

Outline

What About Coordination & Space?

Open problem

- Space-aware coordination medium
- Issues of topology, space and middleware
- Some work already done, space for much more

What About Coordination & Space?

Open problem

- Space-aware coordination medium
- Issues of topology, space and middleware
- Some work already done, space for much more

What About Coordination & Space?

Open problem

- Space-aware coordination medium
- Issues of topology, space and middleware
- Some work already done, space for much more

What About Coordination & Space?

Open problem

- Space-aware coordination medium
- Issues of topology, space and middleware
- Some work already done, space for much more

Outline

Outline

Situatedness & Coordination I

Situatedness. . .

- essentially, strict coupling with the environment
- technically, the ability to properly perceive and react to changes in the environment
- one of the most critical issues in distributed systems
  - conceptual clash between pro-activeness in process behaviour and reactivity w.r.t. environment change
- still one of the most critical issues for artificial intelligence & robotics

Situatedness & Coordination II

. . . & coordination

- essentially, situatedness concerns interaction between processes and the environment
- technically, situatedness can be conceived as a coordination problem
  - how to handle and govern interaction between pro-active processes and an ever-changing environment

Governing interaction

- Intra-system interaction via coordination media as rulers of component-component interaction
- Inter-system interaction via. . . ?
  - coordination media as rulers of component-environment interaction?

Goals

Overall goal of the research

- putting coordination models to test in the challenging context of situatedness
- understanding how classical coordination languages need to be extended to support the coordination of situated processes & distributed systems

Outline

Situating ReSpecT

ReSpecT tuple centres for environment engineering

- Distributed systems are immersed into an environment, and should be reactive to events of *any* sort
- Also, coordination media should mediate any activity toward the environment, allowing for a fruitful interaction

⇒ ReSpecT tuple centres should be able to *capture general environment events*, and to generally *mediate process-environment interaction*

Situating ReSpecT: extensions

- In [Casadei and Omicini, 2009], the ReSpecT language has been revised and extended so as to *capture environment events*, and *express general MAS-environment interactions*

⇒ ReSpecT captures, reacts to, and observes general environment events

⇒ ReSpecT can explicitly interact with the environment

Extending ReSpecT towards Situatedness I

Environment events

- ReSpecT tuple centres are extended to capture two classes of environmental events
  - the interaction with sensors perceiving environmental properties, through *environment predicate* `get`(⟨*Key*⟩,⟨*Value*⟩)
  - the interaction with actuators affecting environmental properties, through *environment predicate* `set`(⟨*Key*⟩,⟨*Value*⟩)
- Source and target of a tuple centre event can be any external resource
  - a suitable identification scheme – both at the syntax and at the infrastructure level – is introduced for environmental resources
- Properties of an environmental event can be observed through the *observation predicate* `env`(⟨*Key*⟩,⟨*Value*⟩)

Extending ReSpecT towards Situatedness II

Environment communication

- The ReSpecT language is extended to express explicit communication with environmental resources
- The body of a ReSpecT reaction can contain a *tuple centre predicate* of the form
  - ⟨*EnvResIdentifier*⟩ `? get(`⟨*Key*⟩,⟨*Value*⟩`)`
    enabling a tuple centre to get properties of environmental resources
  - ⟨*EnvResIdentifier*⟩ `? set(`⟨*Key*⟩,⟨*Value*⟩`)`
    enabling a tuple centre to set properties of environmental resources

Extending ReSpecT towards Situatedness III

Transducers

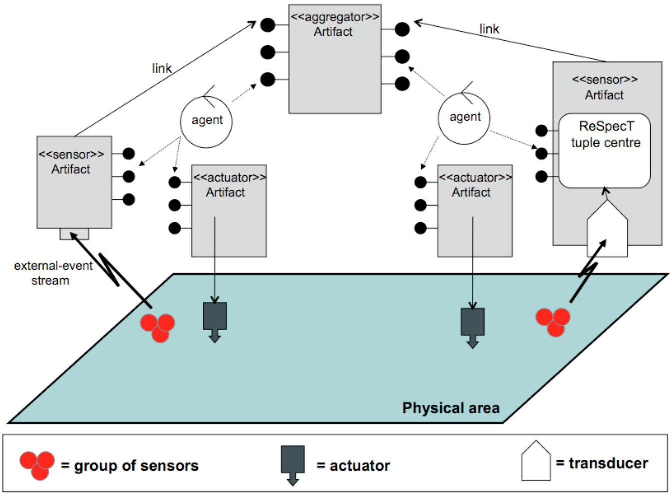
- Specific environment events have to be translated into well-formed ReSpecT tuple centre events
- This should be done at the infrastructure level, through a general-purpose schema that could be specialised according to the nature of any specific resource
- A ReSpecT *transducer* is a component able to bring environment-generated events to a ReSpecT tuple centre (and back), suitably translated according to the general ReSpecT event model
- Each transducer is specialised according to the specific portion of the environment it is in charge of handling—typically, the specific resource it is aimed at handling, like a temperature sensor, or a heater.

Outline

Controlling Environmental Properties of Physical Areas

- A set of real *sensors* are used to measure some environmental property (for instance, temperature) within an area where they are located
- Such information is then exploited to govern suitably placed *actuators* (say, heaters) that can affect the value of the observed property in the environment
- Sensors are supposed to be cheap and non-smart, but provided with some kind of communication interface – either wireless or wired – that makes it possible to send streams of sampled values of the environmental property under observation
- Accordingly, sensors are active devices, that is, devices *pro-actively* sending sensed values at a certain rate with no need of being asked for such data—this is what typically occurs in *pervasive computing* scenarios
- Altogether, actuators and sensors are part of a distributed system aimed at controlling environmental properties (in the case study, temperature), which are affected by actuators based on the values measured by sensors and the designed control policies as well
- Coordination policies can be suitably automated and encapsulated within coordination media working as environment artifacts controlling sensors and actuators

Case Study: ReSpecT-based Architecture



Case Study: Structure of Environment Artifacts

Environment artifacts are built based on of ReSpecT tuple centres:

- `<<sensor>> artifacts` wrapping real temperature sensors which perceive temperature of different areas of the room
- `<<actuator>> artifacts` wrapping actuators, which act as heating devices so as to control temperature
- `<<aggregator>> artifact` provides an aggregated view of the temperature values perceived by sensors spread in the room since it is linked to `<<sensor>> artifacts`:
  - `<<sensor>> artifacts` update tuples on `<<aggregator>> artifact` through *linkability*

Case Study: Sensor Artifacts

```
%(1)
reaction( get(temperature, Temp), from_env, (
     event_time(Time), event_source(sensor(Id)),
     out(sensed_temperature(Id,Temp,Time)),
     tc_aggr@node_aggr ? out(sensed_temperature(Id,Temp)) )
).
%(2)
reaction( out(sensed_temperature(_,Temp,_)), from_tc, (
     in(current_temperature(_)),
     out(current_temperature(Temp)) )
).
```

Behaviour

- Reaction (1) is triggered by external events generated by a temperature sensor
- Reaction (2) updates current temperature

Case Study: Aggregator Artifacts

```
%(4)
reaction( out(sensed_temperature(Id,Temp)), from_tc, (
     in(total_temperature(OldTotalTemp),
     in(sensed_temperature(Id,OldTemp)),
     TotalTemp is OldTotalTemp - OldTemp + Temp,
     out(total_temperature(TotalTemp),
     rd(number_of_sensors(SensorNo),
     AvgTemp is TotalTemp / SensorNo,
     in(average_temp(_)), out(average_temp(AvgTemp)) )
).
```

Behaviour

- Reaction (4) keeps track of the current state of the average temperature

Case Study: Agents

Observable behaviour

Agents are goal-oriented and proactive processes that control temperature of the room

1. get local information from sensor
   `tc_sens@node_i ? rd(current_temperature(Temp_i))`
2. get global information from aggregator
   `tc_aggr@node_aggr ? rd(average_temp(AvgTemp))`
3. deliberate action by determining `TempVar` based on `Temp_i` and `AvgTemp`
4. act upon actuators (if `TempVar` $\neq$ 0)
   `tc-heat_i@node_i ? out(change_temperature(TempVar))`

Case Study: Actuator Artifacts

```
%(3)
reaction( out(change_temperature(TempVar)), from_agent,
   actuator_i ? set(temp_inc,TempVar)
).
```

Behaviour

When the controller agent deliberate an increment in the temperature

- a `tc-heat_i@node_i ? out(change_temperature(TempVar))` reaches the actuator artifact
- by reaction (3), a suitable signal is sent to the actuator, through the suitably-installed transducer

Summing Up

Coordination for Distributed System Engineering

- Engineering the space of interaction among components

Coordination as Governing Interaction

- Enabling vs. Governing

Classes and Features of Coordination Models

- Control-oriented vs. Data-oriented Models

Tuple-based Models

- From LINDA tuple spaces to ReSpecT tuple centres
- Governing distributed systems: from data-oriented to hybrid coordination models
- Time-dependent coordination: experiments of with ReSpecT
- Situated coordination: experiments of with ReSpecT

Summing Up

Coordination for Distributed System Engineering

- Engineering the space of interaction among components

Coordination as Governing Interaction

- Enabling vs. Governing

Classes and Features of Coordination Models

- Control-oriented vs. Data-oriented Models

Tuple-based Models

- From LINDA tuple spaces to ReSpecT tuple centres
- Governing distributed systems: from data-oriented to hybrid coordination models
- Time-dependent coordination: experiments of with ReSpecT
- Situated coordination: experiments of with ReSpecT

Summing Up

Coordination for Distributed System Engineering

- Engineering the space of interaction among components

Coordination as Governing Interaction

- Enabling vs. Governing

Classes and Features of Coordination Models

- Control-oriented vs. Data-oriented Models

Tuple-based Models

- From LINDA tuple spaces to ReSpecT tuple centres
- Governing distributed systems: from data-oriented to hybrid coordination models
- Time-dependent coordination: experiments of with ReSpecT
- Situated coordination: experiments of with ReSpecT

Summing Up

Coordination for Distributed System Engineering

- Engineering the space of interaction among components

Coordination as Governing Interaction

- Enabling vs. Governing

Classes and Features of Coordination Models

- Control-oriented vs. Data-oriented Models

Tuple-based Models

- From LINDA tuple spaces to ReSpecT tuple centres
- Governing distributed systems: from data-oriented to hybrid coordination models
- Time-dependent coordination: experiments of with ReSpecT
- Situated coordination: experiments of with ReSpecT

References I

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

Mathematical Structures in Computer Science, 14:329–366.

Casadei, M. and Omicini, A. (2009).
Situated tuple centres in ReSpecT.
In Shin, S. Y., Ossowski, S., Menezes, R., and Viroli, M., editors, *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, volume III, pages 1361–1368, Honolulu, Hawai'i, USA. ACM.

Ciancarini, P. (1996).
Coordination models and languages as software integrators.
ACM Computing Surveys, 28(2):300–302.

References II

Dastani, M., Arbab, F., and de Boer, F. S. (2005).
Coordination and composition in multi-agent systems.
In Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M. P., and Wooldridge, M. J., editors, *4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 439–446, Utrecht, The Netherlands. ACM.

Dijkstra, E. W. (2002).
Co-operating sequential processes.
In Hansen, P. B., editor, *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, chapter 2, pages 65–138. Springer.
Reprinted. 1st edition: 1965.

References III

Fredriksson, M. and Gustavsson, R. (2004).
Online engineering and open computational systems.
In Bergenti, F., Gleizes, M.-P., and Zambonelli, F., editors, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organization*, pages 377–388. Kluwer Academic Publishers.

Gelernter, D. (1985).
Generative communication in Linda.
ACM Transactions on Programming Languages and Systems, 7(1):80–112.

References IV

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

Goldin, D. Q., Smolka, S. A., and Wegner, P., editors (2006).
Interactive Computation: The New Paradigm.
Springer.

Omicini, A. and Denti, E. (2001).
From tuple spaces to tuple centres.
Science of Computer Programming, 41(3):277–294.

References V

Omicini, A., Ricci, A., and Viroli, M. (2005).
Time-aware coordination in ReSpecT.
In Jacquet, J.-M. and Picco, G. P., editors, *Coordination Models and Languages*, volume 3454 of *LNCS*, pages 268–282. Springer-Verlag.
7th International Conference (COORDINATION 2005), Namur, Belgium, 20–23 April 2005. Proceedings.

Omicini, A., Ricci, A., and Viroli, M. (2007).
Timed environment for Web agents.
Web Intelligence and Agent Systems, 5(2):161–175.

Papadopoulos, G. A. and Arbab, F. (1998).
Coordination models and languages.
In Zelkowitz, M. V., editor, *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press

Coordination-based Systems

Distributed Systems
Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Ingegneria Due
Alma Mater Studiorum—Università di Bologna a Cesena

Academic Year 2011/2012