Components with Symbolic Transition Systems:
a Java Implementation of Rendezvous

Fabrício de Alexandria Fernandes
Jean-Claude Royer Robin Passama

École des Mines de Nantes
Department of Computer Science – OBASCO Group
INRIA Research Centre Rennes - Bretagne Atlantique – LINA

10-07-2007 / CPA 2007

Outline

Outline

Outline

Outline

Outline

Motivation

- Component Based Software Engineering (CBSE)
- Explicit protocols integrated into component interfaces to describe their behaviour in a formal way
- Need of formal analysis methods to analyze component interactions
- Behavioural Interface Description Languages (BIDLs):
    - Architectural analysis and verification issues
    - Relate efficiently design and implementation
- Problem: explicit protocols are often dissociated from component code, not ensured that component execution will respect protocols rules

Motivation

- Component Based Software Engineering (CBSE)
- Explicit protocols integrated into component interfaces to describe their behaviour in a formal way
- Need of formal analysis methods to analyze component interactions
- Behavioural Interface Description Languages (BIDLs):
  - Architectural analysis and verification issues
  - Relate efficiently design and implementation
- Problem: explicit protocols are often dissociated from component code, not ensured that component execution will respect protocols rules

Motivation

- Component Based Software Engineering (CBSE)
- Explicit protocols integrated into component interfaces to describe their behaviour in a formal way
- Need of formal analysis methods to analyze component interactions
- Behavioural Interface Description Languages (BIDLs):
  - Architectural analysis and verification issues
  - Relate efficiently design and implementation
- Problem: explicit protocols are often dissociated from component code, not ensured that component execution will respect protocols rules

Motivation

- Component Based Software Engineering (CBSE)
- Explicit protocols integrated into component interfaces to describe their behaviour in a formal way
- Need of formal analysis methods to analyze component interactions
- Behavioural Interface Description Languages (BIDLs):
  - Architectural analysis and verification issues
  - Relate efficiently design and implementation
- Problem: explicit protocols are often dissociated from component code, not ensured that component execution will respect protocols rules

Motivation

- Component Based Software Engineering (CBSE)
- Explicit protocols integrated into component interfaces to describe their behaviour in a formal way
- Need of formal analysis methods to analyze component interactions
- Behavioural Interface Description Languages (BIDLs):
    - Architectural analysis and verification issues
    - Relate efficiently design and implementation
- Problem: explicit protocols are often dissociated from component code, not ensured that component execution will respect protocols rules

Motivation

- Component Based Software Engineering (CBSE)
- Explicit protocols integrated into component interfaces to describe their behaviour in a formal way
- Need of formal analysis methods to analyze component interactions
- Behavioural Interface Description Languages (BIDLs):
    - Architectural analysis and verification issues
    - Relate efficiently design and implementation
- Problem: explicit protocols are often dissociated from component code, not ensured that component execution will respect protocols rules

Motivation

- Component Based Software Engineering (CBSE)
- Explicit protocols integrated into component interfaces to describe their behaviour in a formal way
- Need of formal analysis methods to analyze component interactions
- Behavioural Interface Description Languages (BIDLs):
  - Architectural analysis and verification issues
  - Relate efficiently design and implementation
- Problem: explicit protocols are often dissociated from component code, not ensured that component execution will respect protocols rules

Our work

- Fill the gap between high-level formal models and implementation of protocols
- Ensure consistency between analysis and execution phases
- Link between specification or design models and programming languages: automated translation of models into programming code
- Long term goal: formal component model with executable protocols which includes associated tools: an STSLib, a formal ADL and analysis tools

Our work

- Fill the gap between high-level formal models and implementation of protocols
- Ensure consistency between analysis and execution phases
- Link between specification or design models and programming languages: automated translation of models into programming code
- Long term goal: formal component model with executable protocols which includes associated tools: an STSLib, a formal ADL and analysis tools

Our work

- Fill the gap between high-level formal models and implementation of protocols
- Ensure consistency between analysis and execution phases
- Link between specification or design models and programming languages: automated translation of models into programming code
- Long term goal: formal component model with executable protocols which includes associated tools: an STSLib, a formal ADL and analysis tools

Our work

- Fill the gap between high-level formal models and implementation of protocols
- Ensure consistency between analysis and execution phases
- Link between specification or design models and programming languages: automated translation of models into programming code
- Long term goal: formal component model with executable protocols which includes associated tools: an STSLib, a formal ADL and analysis tools

STS-Oriented Component Model

- Subset of Korrigan model [Poizat and Royer JUCS06] based on ADL ontology: configurations (architectures) made of components with ports and connections
- Two types of components
  - Primitive: based on STS, to be presented in the next slides
  - Composite: reusable compositions of components (i.e. architectures)
- A glue notation to define communications, currently restricted to n-ary communications with one emitter and several receivers

STS-Oriented Component Model

- Subset of Korrigan model [Poizat and Royer JUCS06] based on ADL ontology: configurations (architectures) made of components with ports and connections
- Two types of components
  - Primitive: based on STS, to be presented in the next slides
  - Composite: reusable compositions of components (i.e. architectures)
- A glue notation to define communications, currently restricted to n-ary communications with one emitter and several receivers

STS-Oriented Component Model

- Subset of Korrigan model [Poizat and Royer JUCS06] based on ADL ontology: configurations (architectures) made of components with ports and connections
- Two types of components
  - Primitive: based on STS, to be presented in the next slides
  - Composite: reusable compositions of components (i.e. architectures)
- A glue notation to define communications, currently restricted to n-ary communications with one emitter and several receivers

STS-Oriented Component Model

- Subset of Korrigan model [Poizat and Royer JUCS06] based on ADL ontology: configurations (architectures) made of components with ports and connections
- Two types of components
    - Primitive: based on STS, to be presented in the next slides
    - Composite: reusable compositions of components (i.e. architectures)
- A glue notation to define communications, currently restricted to n-ary communications with one emitter and several receivers

STS-Oriented Component Model

- Subset of Korrigan model [Poizat and Royer JUCS06] based on ADL ontology: configurations (architectures) made of components with ports and connections
- Two types of components
    - Primitive: based on STS, to be presented in the next slides
    - Composite: reusable compositions of components (i.e. architectures)
- A glue notation to define communications, currently restricted to n-ary communications with one emitter and several receivers

STS Model

- Primitive component made of ports and a protocol described in the STS formalism
- STS: states + transitions between states
- STS transition general syntax: [guard] event/action
    - guard: condition to trigger the transition
    - event: dynamic event possibly with emission ! or receipt ? (notification of the action execution)
    - action: action to be performed
    - Action may be described in an algebraic or a programming style
    - A Java translation from axiom description has been experimented

STS Model

- Primitive component made of ports and a protocol described in the STS formalism
- STS: states + transitions between states
- STS transition general syntax: [guard] event/action
    - guard: condition to trigger the transition
    - event: dynamic event possibly with emission ! or receipt ? (notification of the action execution)
    - action: action to be performed
    - Action may be described in an algebraic or a programming style
    - A Java translation from axiom description has been experimented

STS Model

- Primitive component made of ports and a protocol described in the STS formalism
- STS: states + transitions between states
- STS transition general syntax: [guard] event/action
  - guard: condition to trigger the transition
  - event: dynamic event possibly with emission ! or receipt ? (notification of the action execution)
  - action: action to be performed
  - Action may be described in an algebraic or a programming style
  - A Java translation from axiom description has been experimented

STS Model

- Primitive component made of ports and a protocol described in the STS formalism
- STS: states + transitions between states
- STS transition general syntax: [guard] event/action
    - guard: condition to trigger the transition
    - event: dynamic event possibly with emission ! or receipt ? (notification of the action execution)
    - action: action to be performed
    - Action may be described in an algebraic or a programming style
    - A Java translation from axiom description has been experimented

STS Model

- Primitive component made of ports and a protocol described in the STS formalism
- STS: states + transitions between states
- STS transition general syntax: [guard] event/action
  - guard: condition to trigger the transition
  - event: dynamic event possibly with emission ! or receipt ? (notification of the action execution)
  - action: action to be performed
  - Action may be described in an algebraic or a programming style
  - A Java translation from axiom description has been experimented

STS Model

- Primitive component made of ports and a protocol described in the STS formalism
- STS: states + transitions between states
- STS transition general syntax: [guard] event/action
    - guard: condition to trigger the transition
    - event: dynamic event possibly with emission ! or receipt ? (notification of the action execution)
    - action: action to be performed
    - Action may be described in an algebraic or a programming style
    - A Java translation from axiom description has been experimented

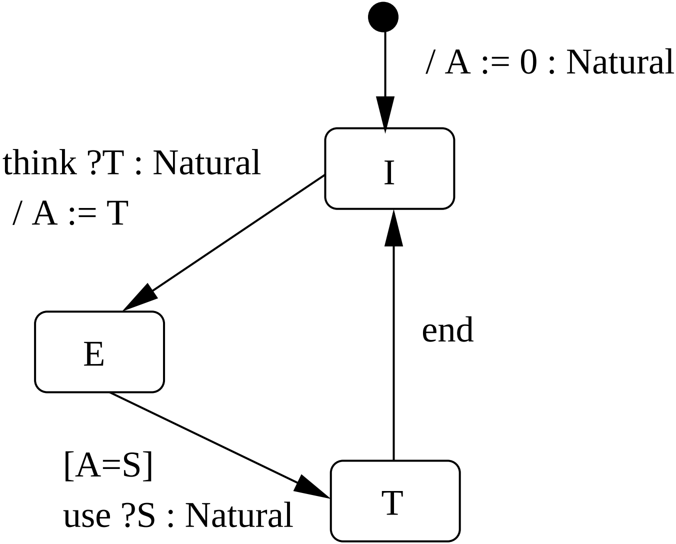
STS Model

- Primitive component made of ports and a protocol described in the STS formalism
- STS: states + transitions between states
- STS transition general syntax: [guard] event/action
    - guard: condition to trigger the transition
    - event: dynamic event possibly with emission ! or receipt ? (notification of the action execution)
    - action: action to be performed
    - Action may be described in an algebraic or a programming style
    - A Java translation from axiom description has been experimented

STS Model

- Primitive component made of ports and a protocol described in the STS formalism
- STS: states + transitions between states
- STS transition general syntax: [guard] event/action
  - guard: condition to trigger the transition
  - event: dynamic event possibly with emission ! or receipt ? (notification of the action execution)
  - action: action to be performed
  - Action may be described in an algebraic or a programming style
  - A Java translation from axiom description has been experimented

Example of STS Component: The Process



Architecture and Composite

- Note: We are rather discussing component types rather than a component instance
- An architecture is a closed composite, that is not designed to interact with the outside
- A composite is an assembly of primitive and composite components
- Ports: connection points that externalizes the triggering of a given event in the STS protocol
- Connections: primitive bindings between ports
- Connected ports: synchronization of corresponding events

Architecture and Composite

- Note: We are rather discussing component types rather than a component instance
- An architecture is a closed composite, that is not designed to interact with the outside
- A composite is an assembly of primitive and composite components
- Ports: connection points that externalizes the triggering of a given event in the STS protocol
- Connections: primitive bindings between ports
- Connected ports: synchronization of corresponding events

Architecture and Composite

- Note: We are rather discussing component types rather than a component instance
- An architecture is a closed composite, that is not designed to interact with the outside
- A composite is an assembly of primitive and composite components
- Ports: connection points that externalizes the triggering of a given event in the STS protocol
- Connections: primitive bindings between ports
- Connected ports: synchronization of corresponding events

Architecture and Composite

- Note: We are rather discussing component types rather than a component instance
- An architecture is a closed composite, that is not designed to interact with the outside
- A composite is an assembly of primitive and composite components
- Ports: connection points that externalizes the triggering of a given event in the STS protocol
- Connections: primitive bindings between ports
- Connected ports: synchronization of corresponding events

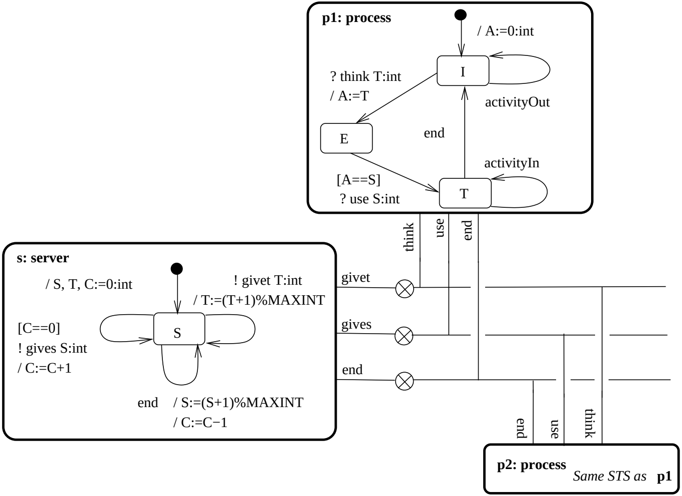
Architecture and Composite

- Note: We are rather discussing component types rather than a component instance
- An architecture is a closed composite, that is not designed to interact with the outside
- A composite is an assembly of primitive and composite components
- Ports: connection points that externalizes the triggering of a given event in the STS protocol
- Connections: primitive bindings between ports
- Connected ports: synchronization of corresponding events

Architecture and Composite

- Note: We are rather discussing component types rather than a component instance
- An architecture is a closed composite, that is not designed to interact with the outside
- A composite is an assembly of primitive and composite components
- Ports: connection points that externalizes the triggering of a given event in the STS protocol
- Connections: primitive bindings between ports
- Connected ports: synchronization of corresponding events

Architecture Example



Rendezvous Principle

- Synchronization of several events: triggering them in any real order but in the same logical time
- With communication: sender necessarily initiates a value computation and communicate it to the receivers
- Primitive components involved in synchronization cannot trigger any other event during this synchronization
- Provides execution actions of all the participants and 1 to n communications

Rendezvous Principle

- Synchronization of several events: triggering them in any real order but in the same logical time
- With communication: sender necessarily initiates a value computation and communicate it to the receivers
- Primitive components involved in synchronization cannot trigger any other event during this synchronization
- Provides execution actions of all the participants and 1 to n communications

Rendezvous Principle

- Synchronization of several events: triggering them in any real order but in the same logical time
- With communication: sender necessarily initiates a value computation and communicate it to the receivers
- Primitive components involved in synchronization cannot trigger any other event during this synchronization
- Provides execution actions of all the participants and 1 to n communications

Rendezvous Principle

- Synchronization of several events: triggering them in any real order but in the same logical time
- With communication: sender necessarily initiates a value computation and communicate it to the receivers
- Primitive components involved in synchronization cannot trigger any other event during this synchronization
- Provides execution actions of all the participants and 1 to n communications

Three Modes for Components Interaction

- Asynchronous activity: one component executes an action independently (no interaction)
- Rendezvous without communication: n components execute a given action in the same logical time
- Rendezvous: latter case + a component emits a value and other receives it during the rendezvous. Receiver guards check the emitted value (guards with receipt)

Three Modes for Components Interaction

- Asynchronous activity: one component executes an action independently (no interaction)
- Rendezvous without communication: n components execute a given action in the same logical time
- Rendezvous: latter case + a component emits a value and other receives it during the rendezvous. Receiver guards check the emitted value (guards with receipt)

Three Modes for Components Interaction

- Asynchronous activity: one component executes an action independently (no interaction)
- Rendezvous without communication: n components execute a given action in the same logical time
- Rendezvous: latter case + a component emits a value and other receives it during the rendezvous. Receiver guards check the emitted value (guards with receipt)

Synchronization Vectors

- Concept coming from the synchronous product of automata
- Definition: vector of events that denotes a possible synchronization at runtime between a set of events
- Computed according to the connections between component ports
- Defined according to an arbitrary ordering of primitive components
- The connections define a computation of synchronization vectors, for instance ⊕ communication
- Also useful for configuring runtime support of components

Synchronization Vectors

- Concept coming from the synchronous product of automata
- Definition: vector of events that denotes a possible synchronization at runtime between a set of events
- Computed according to the connections between component ports
- Defined according to an arbitrary ordering of primitive components
- The connections define a computation of synchronization vectors, for instance ⊕ communication
- Also useful for configuring runtime support of components

Synchronization Vectors

- Concept coming from the synchronous product of automata
- Definition: vector of events that denotes a possible synchronization at runtime between a set of events
- Computed according to the connections between component ports
- Defined according to an arbitrary ordering of primitive components
- The connections define a computation of synchronization vectors, for instance $\oplus$ communication
- Also useful for configuring runtime support of components

Synchronization Vectors

- Concept coming from the synchronous product of automata
- Definition: vector of events that denotes a possible synchronization at runtime between a set of events
- Computed according to the connections between component ports
- Defined according to an arbitrary ordering of primitive components
- The connections define a computation of synchronization vectors, for instance $\oplus$ communication
- Also useful for configuring runtime support of components

Synchronization Vectors

- Concept coming from the synchronous product of automata
- Definition: vector of events that denotes a possible synchronization at runtime between a set of events
- Computed according to the connections between component ports
- Defined according to an arbitrary ordering of primitive components
- The connections define a computation of synchronization vectors, for instance $\oplus$ communication
- Also useful for configuring runtime support of components

Synchronization Vectors

- Concept coming from the synchronous product of automata
- Definition: vector of events that denotes a possible synchronization at runtime between a set of events
- Computed according to the connections between component ports
- Defined according to an arbitrary ordering of primitive components
- The connections define a computation of synchronization vectors, for instance $\oplus$ communication
- Also useful for configuring runtime support of components

Guard with Receipt

- Important construction with a specific semantics: components can conditionally receive and synchronize on a value in the same logical time
- Benefit: to increase the abstraction and reduce the size of finite state machine
- Example guard with receipt and no action: [A=S] ? use S:int .
- Three steps: receipt, guard checking, (null) action
- Rendezvous: all three steps have to be synchronous
- Major implementation issue: keep the model semantic and components execution consistent

Guard with Receipt

- Important construction with a specific semantics: components can conditionally receive and synchronize on a value in the same logical time
- Benefit: to increase the abstraction and reduce the size of finite state machine
- Example guard with receipt and no action: [A=S] ? use S:int .
- Three steps: receipt, guard checking, (null) action
- Rendezvous: all three steps have to be synchronous
- Major implementation issue: keep the model semantic and components execution consistent

Guard with Receipt

- Important construction with a specific semantics: components can conditionally receive and synchronize on a value in the same logical time
- Benefit: to increase the abstraction and reduce the size of finite state machine
- Example guard with receipt and no action: [A=S] ? use S:int .
- Three steps: receipt, guard checking, (null) action
- Rendezvous: all three steps have to be synchronous
- Major implementation issue: keep the model semantic and components execution consistent

Guard with Receipt

- Important construction with a specific semantics: components can conditionally receive and synchronize on a value in the same logical time
- Benefit: to increase the abstraction and reduce the size of finite state machine
- Example guard with receipt and no action: [A=S] ? use S:int .
- Three steps: receipt, guard checking, (null) action
- Rendezvous: all three steps have to be synchronous
- Major implementation issue: keep the model semantic and components execution consistent

Guard with Receipt

- Important construction with a specific semantics: components can conditionally receive and synchronize on a value in the same logical time
- Benefit: to increase the abstraction and reduce the size of finite state machine
- Example guard with receipt and no action: [A=S] ? use S:int .
- Three steps: receipt, guard checking, (null) action
- Rendezvous: all three steps have to be synchronous
- Major implementation issue: keep the model semantic and components execution consistent

Guard with Receipt

- Important construction with a specific semantics: components can conditionally receive and synchronize on a value in the same logical time
- Benefit: to increase the abstraction and reduce the size of finite state machine
- Example guard with receipt and no action: [A=S] ? use S:int .
- Three steps: receipt, guard checking, (null) action
- Rendezvous: all three steps have to be synchronous
- Major implementation issue: keep the model semantic and components execution consistent

Implementation Overview

- Implementing STS requires to manage different development steps:
    - Implementing the data part
    - Representing the protocol
    - Gluing the data part and the protocol into a primitive component (intra-component composition)
    - Implementing components synchronization and communication (inter-component composition)

Implementation Overview

- Implementing STS requires to manage different development steps:
  - Implementing the data part
  - Representing the protocol
  - Gluing the data part and the protocol into a primitive component (intra-component composition)
  - Implementing components synchronization and communication (inter-component composition)

Implementation Overview

- Implementing STS requires to manage different development steps:
  - Implementing the data part
  - Representing the protocol
  - Gluing the data part and the protocol into a primitive component (intra-component composition)
  - Implementing components synchronization and communication (inter-component composition)

Implementation Overview

- Implementing STS requires to manage different development steps:
  - Implementing the data part
  - Representing the protocol
  - Gluing the data part and the protocol into a primitive component (intra-component composition)
  - Implementing components synchronization and communication (inter-component composition)

Implementation Overview

- Implementing STS requires to manage different development steps:
  - Implementing the data part
  - Representing the protocol
  - Gluing the data part and the protocol into a primitive component (intra-component composition)
  - Implementing components synchronization and communication (inter-component composition)

Implementation of the STS

- Separation of the FSM notations to the data part: simplifies the implementation and promotes the reuse
- *Dynamic part*: states, transitions and some names (guards, events, receipt variables, senders and actions)
- *Data part*: Java class implementing the formal data part with a real implementation of the names with methods
- Both parts glued thanks to a normalized Java interface
- *Emitter*: pure function computing the emitted value in a given state of the component
- *Guard*: boolean function implementing a condition
- A receiver is implemented as a method with parameters

Implementation of the STS

- Separation of the FSM notations to the data part: simplifies the implementation and promotes the reuse
- *Dynamic part*: states, transitions and some names (guards, events, receipt variables, senders and actions)
- *Data part*: Java class implementing the formal data part with a real implementation of the names with methods
- Both parts glued thanks to a normalized Java interface
- *Emitter*: pure function computing the emitted value in a given state of the component
- *Guard*: boolean function implementing a condition
- A receiver is implemented as a method with parameters

Implementation of the STS

- Separation of the FSM notations to the data part: simplifies the implementation and promotes the reuse
- *Dynamic part*: states, transitions and some names (guards, events, receipt variables, senders and actions)
- *Data part*: Java class implementing the formal data part with a real implementation of the names with methods
- Both parts glued thanks to a normalized Java interface
- *Emitter*: pure function computing the emitted value in a given state of the component
- *Guard*: boolean function implementing a condition
- A receiver is implemented as a method with parameters

Implementation of the STS

- Separation of the FSM notations to the data part: simplifies the implementation and promotes the reuse
- *Dynamic part*: states, transitions and some names (guards, events, receipt variables, senders and actions)
- *Data part*: Java class implementing the formal data part with a real implementation of the names with methods
- Both parts glued thanks to a normalized Java interface
- *Emitter*: pure function computing the emitted value in a given state of the component
- *Guard*: boolean function implementing a condition
- A receiver is implemented as a method with parameters

Implementation of the STS

- Separation of the FSM notations to the data part: simplifies the implementation and promotes the reuse
- *Dynamic part*: states, transitions and some names (guards, events, receipt variables, senders and actions)
- *Data part*: Java class implementing the formal data part with a real implementation of the names with methods
- Both parts glued thanks to a normalized Java interface
- *Emitter*: pure function computing the emitted value in a given state of the component
- *Guard*: boolean function implementing a condition
- A receiver is implemented as a method with parameters

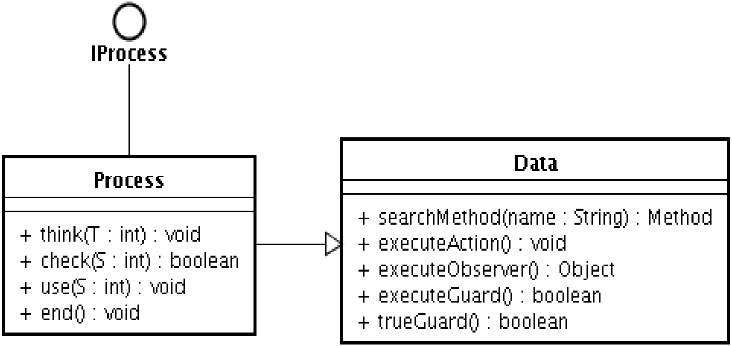
Implementation of the STS

- Separation of the FSM notations to the data part: simplifies the implementation and promotes the reuse
- *Dynamic part*: states, transitions and some names (guards, events, receipt variables, senders and actions)
- *Data part*: Java class implementing the formal data part with a real implementation of the names with methods
- Both parts glued thanks to a normalized Java interface
- *Emitter*: pure function computing the emitted value in a given state of the component
- *Guard*: boolean function implementing a condition
- A receiver is implemented as a method with parameters

Implementation of the STS

- Separation of the FSM notations to the data part: simplifies the implementation and promotes the reuse
- *Dynamic part*: states, transitions and some names (guards, events, receipt variables, senders and actions)
- *Data part*: Java class implementing the formal data part with a real implementation of the names with methods
- Both parts glued thanks to a normalized Java interface
- *Emitter*: pure function computing the emitted value in a given state of the component
- *Guard*: boolean function implementing a condition
- A receiver is implemented as a method with parameters

STS Implementation Schema



STS Implementation Principles

- Combination of a protocol and existing Java code data part (passive class that implements an interface)
- Implemented with an active object (thread in Java) to execute STS protocol and to call the passive object
- An STS defines events, guards, emitters and actions related to the Java interface of the data part class
- Automatic generation from STS to Java skeleton

STS Implementation Principles

- Combination of a protocol and existing Java code data part (passive class that implements an interface)
- Implemented with an active object (thread in Java) to execute STS protocol and to call the passive object
- An STS defines events, guards, emitters and actions related to the Java interface of the data part class
- Automatic generation from STS to Java skeleton

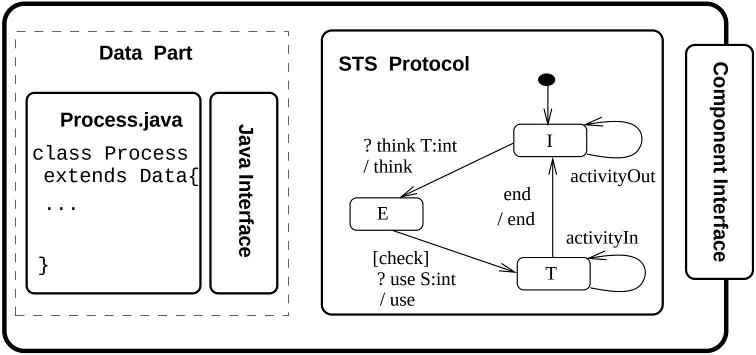
STS Implementation Principles

- Combination of a protocol and existing Java code data part (passive class that implements an interface)
- Implemented with an active object (thread in Java) to execute STS protocol and to call the passive object
- An STS defines events, guards, emitters and actions related to the Java interface of the data part class
- Automatic generation from STS to Java skeleton

STS Implementation Principles

- Combination of a protocol and existing Java code data part (passive class that implements an interface)
- Implemented with an active object (thread in Java) to execute STS protocol and to call the passive object
- An STS defines events, guards, emitters and actions related to the Java interface of the data part class
- Automatic generation from STS to Java skeleton

Implementation of the Process Primitive Component



Data Part

Process.java

```
class Process
 extends Data{
 ...

}
```

Java Interface

STS Protocol

Component Interface

Rules to Generate Interfaces

- Translation rules for one emission and one receipt

[guard] event !emitter:Type / action

```
public boolean guard();
public Type emitter();
public void action(Type var);
```

[guard] event ?var:Type / action

```
public boolean guard(Type var)
public void action(Type var);
```

- Automatic generation from STS to Java skeleton

```
public interface IProcess {
  public void think (int T);
  public boolean check (int S);  // check for guard (A == S)
  public void use (int S);
  public void end ();
}
```

Rules to Generate Interfaces

- Translation rules for one emission and one receipt

[guard] event !emitter:Type / action

```
public boolean guard();
public Type emitter();
public void action(Type var);
```

[guard] event ?var:Type / action

```
public boolean guard(Type var
public void action(Type var);
```

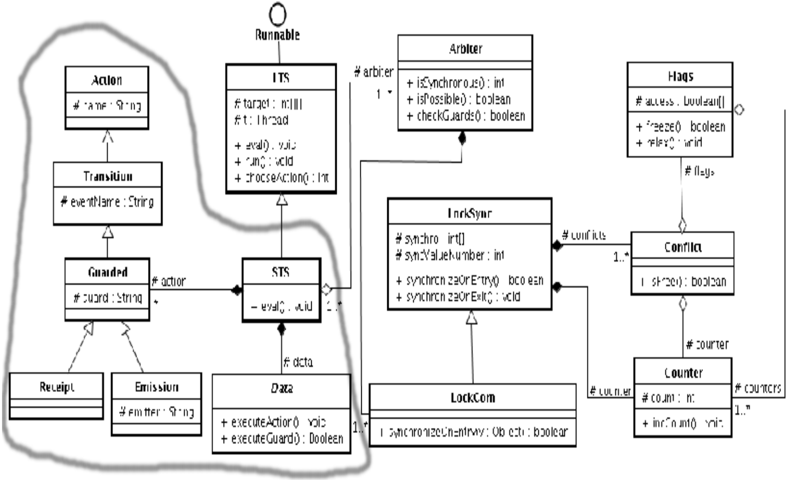
- Automatic generation from STS to Java skeleton

```
public interface IProcess {
  public void think (int T);
  public boolean check (int S);  // check for guard (A == S)
  public void use (int S);
  public void end ();
}
```

Java Class for the Process STS

```
public class Process extends Data implements IProcess {

  protected int A;

  public Process () {
    this.A = 0;
  }

  public void think (int T) {
    this.A = T;
  }

  // guard with receipt
  public boolean check (int S) {
    return this.A == S;
  }

  // use action with receipt
  public void use (int S) {
    System.out.println ("Enter␣critical␣section");
  }

  public void end () {
    System.out.println ("Leaving␣critical␣section");
  }

}
```

Partial UML Class Diagram



Implementation of Concurrent Composition

- Input: several STSs and synchronization vectors that bind their events
- Configures STS runtime support that conforms to the semantic model
- Consequences:
  - Each STS has its own execution thread
  - All STSs have to be synchronized depending on synchronization vectors.
- Primitive component (at runtime): unique thread
- Composite component: collection of interacting threads
- Synchronization of threads: supported by a specific rendezvous mechanism

Implementation of Concurrent Composition

- Input: several STSs and synchronization vectors that bind their events
- Configures STS runtime support that conforms to the semantic model
- Consequences:
    - Each STS has its own execution thread
    - All STSs have to be synchronized depending on synchronization vectors.
- Primitive component (at runtime): unique thread
- Composite component: collection of interacting threads
- Synchronization of threads: supported by a specific rendezvous mechanism

Implementation of Concurrent Composition

- Input: several STSs and synchronization vectors that bind their events
- Configures STS runtime support that conforms to the semantic model
- Consequences:
    - Each STS has its own execution thread
    - All STSs have to be synchronized depending on synchronization vectors.
- Primitive component (at runtime): unique thread
- Composite component: collection of interacting threads
- Synchronization of threads: supported by a specific rendezvous mechanism

Implementation of Concurrent Composition

- Input: several STSs and synchronization vectors that bind their events
- Configures STS runtime support that conforms to the semantic model
- Consequences:
  - Each STS has its own execution thread
  - All STSs have to be synchronized depending on synchronization vectors.
- Primitive component (at runtime): unique thread
- Composite component: collection of interacting threads
- Synchronization of threads: supported by a specific rendezvous mechanism

Implementation of Concurrent Composition

- Input: several STSs and synchronization vectors that bind their events
- Configures STS runtime support that conforms to the semantic model
- Consequences:
    - Each STS has its own execution thread
    - All STSs have to be synchronized depending on synchronization vectors.
- Primitive component (at runtime): unique thread
- Composite component: collection of interacting threads
- Synchronization of threads: supported by a specific rendezvous mechanism

Implementation of Concurrent Composition

- Input: several STSs and synchronization vectors that bind their events
- Configures STS runtime support that conforms to the semantic model
- Consequences:
  - Each STS has its own execution thread
  - All STSs have to be synchronized depending on synchronization vectors.
- Primitive component (at runtime): unique thread
- Composite component: collection of interacting threads
- Synchronization of threads: supported by a specific rendezvous mechanism

Implementation of Concurrent Composition

- Input: several STSs and synchronization vectors that bind their events
- Configures STS runtime support that conforms to the semantic model
- Consequences:
  - Each STS has its own execution thread
  - All STSs have to be synchronized depending on synchronization vectors.
- Primitive component (at runtime): unique thread
- Composite component: collection of interacting threads
- Synchronization of threads: supported by a specific rendezvous mechanism

Implementation of Concurrent Composition

- Input: several STSs and synchronization vectors that bind their events
- Configures STS runtime support that conforms to the semantic model
- Consequences:
  - Each STS has its own execution thread
  - All STSs have to be synchronized depending on synchronization vectors.
- Primitive component (at runtime): unique thread
- Composite component: collection of interacting threads
- Synchronization of threads: supported by a specific rendezvous mechanism

Basic Barrier Principles

- Started with a mechanism to implement the synchronization of LTSs [Noyé et al, GPCE06]
- Synchronization possible between two actions with the same name
- An arbiter controls that synchronizations are correctly handled
- Two synchronization barriers with a Java monitor: one barrier to enter and other one to leave
- Why two? With only one, asynchronous actions may be triggered at the same logical time (inconsistent)

Basic Barrier Principles

- Started with a mechanism to implement the synchronization of LTSs [Noyé et al, GPCE06]
- Synchronization possible between two actions with the same name
- An arbiter controls that synchronizations are correctly handled
- Two synchronization barriers with a Java monitor: one barrier to enter and other one to leave
- Why two? With only one, asynchronous actions may be triggered at the same logical time (inconsistent)

Basic Barrier Principles

- Started with a mechanism to implement the synchronization of LTSs [Noyé et al, GPCE06]
- Synchronization possible between two actions with the same name
- An arbiter controls that synchronizations are correctly handled
- Two synchronization barriers with a Java monitor: one barrier to enter and other one to leave
- Why two? With only one, asynchronous actions may be triggered at the same logical time (inconsistent)

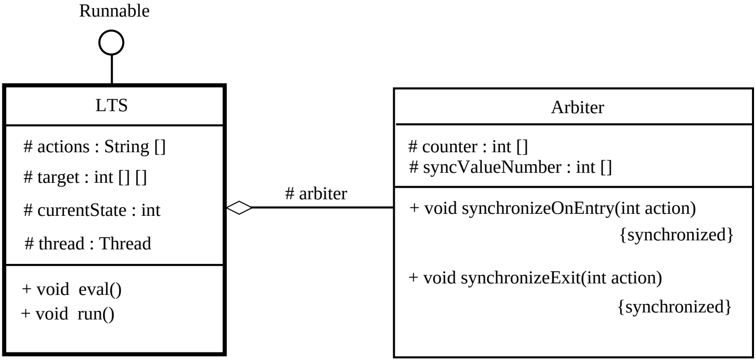
Basic Barrier Principles

- Started with a mechanism to implement the synchronization of LTSs [Noyé et al, GPCE06]
- Synchronization possible between two actions with the same name
- An arbiter controls that synchronizations are correctly handled
- Two synchronization barriers with a Java monitor: one barrier to enter and other one to leave
- Why two? With only one, asynchronous actions may be triggered at the same logical time (inconsistent)

Basic Barrier Principles

- Started with a mechanism to implement the synchronization of LTSs [Noyé et al, GPCE06]
- Synchronization possible between two actions with the same name
- An arbiter controls that synchronizations are correctly handled
- Two synchronization barriers with a Java monitor: one barrier to enter and other one to leave
- Why two? With only one, asynchronous actions may be triggered at the same logical time (inconsistent)

Basic Barrier Diagram



Runnable

LTS

\# actions : String []

\# target : int [] []

\# currentState : int

\# thread : Thread

\+ void eval()

\+ void run()

\# arbiter

Arbiter

\# counter : int []

\# syncValueNumber : int []

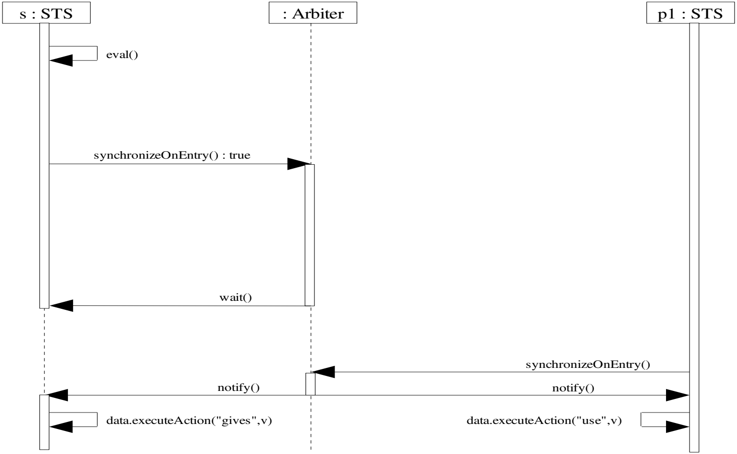
\+ void synchronizeOnEntry(int action) {synchronized}

\+ void synchronizeExit(int action) {synchronized}

Synchronization Barrier

```
synchronized public void synchronizeOnEntry (int action) {
  if (counter[action] < syncValueNumber[action] – 1) {
    counter[action]++;                    // we are not the last thread
    try {                                 // so block
      wait ();
    } catch (InterruptedException e) {}
  } else {
    counter[action]=0;                    // we are the last thread
    notifyAll ();                         // so wake up all
  }
}
```

Sequence Entering the Barrier



Synchronization Vector Representation

- First improvement: relax the restriction on names for synchronization (reuse purposes)
- Solution: set of synchronizations vectors each one represents a possible synchronization between some events
- Event and action name associated inside a Transition
- Representation by a new class `LockSync` with the barrier methods
- Method `isSynchronous` to choose one `LockSync` object

Synchronization Vector Representation

- First improvement: relax the restriction on names for synchronization (reuse purposes)
- Solution: set of synchronizations vectors each one represents a possible synchronization between some events
- Event and action name associated inside a Transition
- Representation by a new class `LockSync` with the barrier methods
- Method `isSynchronous` to choose one `LockSync` object

Synchronization Vector Representation

- First improvement: relax the restriction on names for synchronization (reuse purposes)
- Solution: set of synchronizations vectors each one represents a possible synchronization between some events
- Event and action name associated inside a Transition
- Representation by a new class `LockSync` with the barrier methods
- Method `isSynchronous` to choose one `LockSync` object

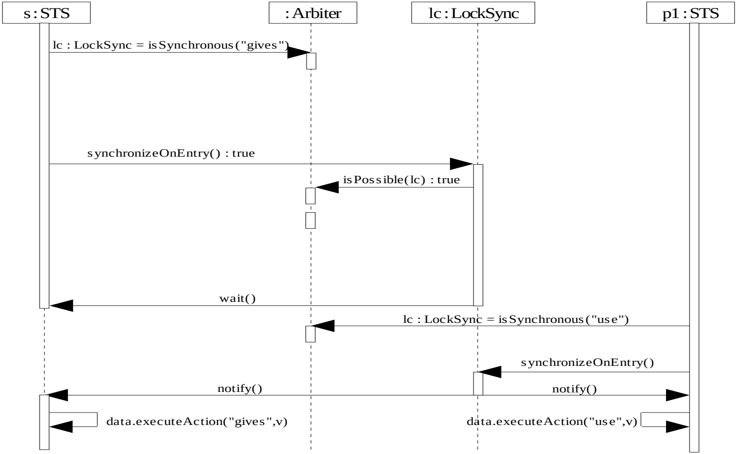
Synchronization Vector Representation

- First improvement: relax the restriction on names for synchronization (reuse purposes)
- Solution: set of synchronizations vectors each one represents a possible synchronization between some events
- Event and action name associated inside a Transition
- Representation by a new class `LockSync` with the barrier methods
- Method `isSynchronous` to choose one `LockSync` object

Synchronization Vector Representation

- First improvement: relax the restriction on names for synchronization (reuse purposes)
- Solution: set of synchronizations vectors each one represents a possible synchronization between some events
- Event and action name associated inside a Transition
- Representation by a new class `LockSync` with the barrier methods
- Method `isSynchronous` to choose one `LockSync` object

Sequence Entering the Barrier



Independent Synchronizations

- Problem: synchronization serialized (single arbiter and entry/exit methods are synchronized)
- Solution: LockSync class
- Independent synchronization: one from another iff it does not belong to its conflict set (Conflict class)
- Conflict of a synchronization: defined as set of synchronizations which synchronize on a common component
- On the example, synchronizations are mutually conflicting because of the central server component

Independent Synchronizations

- Problem: synchronization serialized (single arbiter and entry/exit methods are synchronized)
- Solution: LockSync class
- Independent synchronization: one from another iff it does not belong to its conflict set (Conflict class)
- Conflict of a synchronization: defined as set of synchronizations which synchronize on a common component
- On the example, synchronizations are mutually conflicting because of the central server component

Independent Synchronizations

- Problem: synchronization serialized (single arbiter and entry/exit methods are synchronized)
- Solution: LockSync class
- Independent synchronization: one from another iff it does not belong to its conflict set (Conflict class)
- Conflict of a synchronization: defined as set of synchronizations which synchronize on a common component
- On the example, synchronizations are mutually conflicting because of the central server component

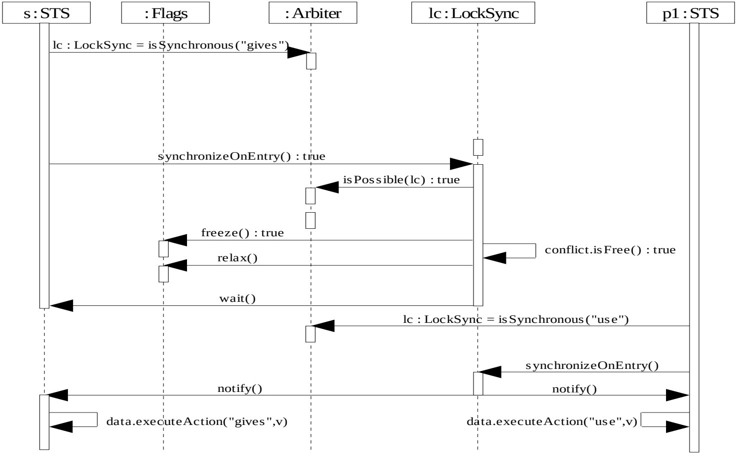
Independent Synchronizations

- Problem: synchronization serialized (single arbiter and entry/exit methods are synchronized)
- Solution: LockSync class
- Independent synchronization: one from another iff it does not belong to its conflict set (Conflict class)
- Conflict of a synchronization: defined as set of synchronizations which synchronize on a common component
- On the example, synchronizations are mutually conflicting because of the central server component

Independent Synchronizations

- Problem: synchronization serialized (single arbiter and entry/exit methods are synchronized)
- Solution: LockSync class
- Independent synchronization: one from another iff it does not belong to its conflict set (Conflict class)
- Conflict of a synchronization: defined as set of synchronizations which synchronize on a common component
- On the example, synchronizations are mutually conflicting because of the central server component

Sequence Entering the Barrier



s : STS

: Flags

: Arbiter

lc : LockSync

p1 : STS

lc : LockSync = isSynchronous("gives")

synchronizeOnEntry() : true

isPossible(lc) : true

freeze() : true

conflict.isFree() : true

relax()

wait()

lc : LockSync = isSynchronous("use")

synchronizeOnEntry()

notify()

notify()

data.executeAction("gives",v)

data.executeAction("use",v)

Guards with Communication

- More complex STS transitions: addition of the classes Guarded, Emission and Receipt
- Abstract class Data: execution of guards, emitters and actions on an instance
- `eval` method modified to manage synchronous actions with communication
- Introduction of the class `LockCom` (specialization of LockSync with the communication case)
- New methods: `setEmittedValue` to communicate the values to the LockSync objects; `checkGuards` to verify if the guards are true; `eval` modified to retrieve the communicated values

Guards with Communication

- More complex STS transitions: addition of the classes Guarded, Emission and Receipt
- Abstract class Data: execution of guards, emitters and actions on an instance
- `eval` method modified to manage synchronous actions with communication
- Introduction of the class `LockCom` (specialization of LockSync with the communication case)
- New methods: `setEmittedValue` to communicate the values to the LockSync objects; `checkGuards` to verify if the guards are true; `eval` modified to retrieve the communicated values

Guards with Communication

- More complex STS transitions: addition of the classes Guarded, Emission and Receipt
- Abstract class Data: execution of guards, emitters and actions on an instance
- `eval` method modified to manage synchronous actions with communication
- Introduction of the class `LockCom` (specialization of LockSync with the communication case)
- New methods: `setEmittedValue` to communicate the values to the LockSync objects; `checkGuards` to verify if the guards are true; `eval` modified to retrieve the communicated values

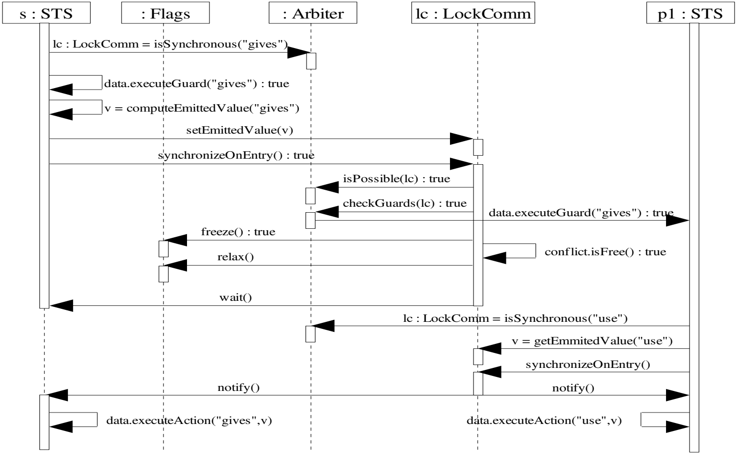
Guards with Communication

- More complex STS transitions: addition of the classes Guarded, Emission and Receipt
- Abstract class Data: execution of guards, emitters and actions on an instance
- `eval` method modified to manage synchronous actions with communication
- Introduction of the class `LockCom` (specialization of LockSync with the communication case)
- New methods: `setEmittedValue` to communicate the values to the LockSync objects; `checkGuards` to verify if the guards are true; `eval` modified to retrieve the communicated values

Guards with Communication

- More complex STS transitions: addition of the classes Guarded, Emission and Receipt
- Abstract class Data: execution of guards, emitters and actions on an instance
- `eval` method modified to manage synchronous actions with communication
- Introduction of the class `LockCom` (specialization of LockSync with the communication case)
- New methods: `setEmittedValue` to communicate the values to the LockSync objects; `checkGuards` to verify if the guards are true; `eval` modified to retrieve the communicated values

Sequence Entering the Barrier



s : STS

: Flags

: Arbiter

lc : LockComm

p1 : STS

lc : LockComm = isSynchronous("gives")

data.executeGuard("gives") : true

v = computeEmittedValue("gives")

setEmittedValue(v)

synchronizeOnEntry() : true

isPossible(lc) : true

checkGuards(lc) : true

data.executeGuard("gives") : true

freeze() : true

conflict.isFree() : true

relax()

wait()

lc : LockComm = isSynchronous("use")

v = getEmmitedValue("use")

synchronizeOnEntry()

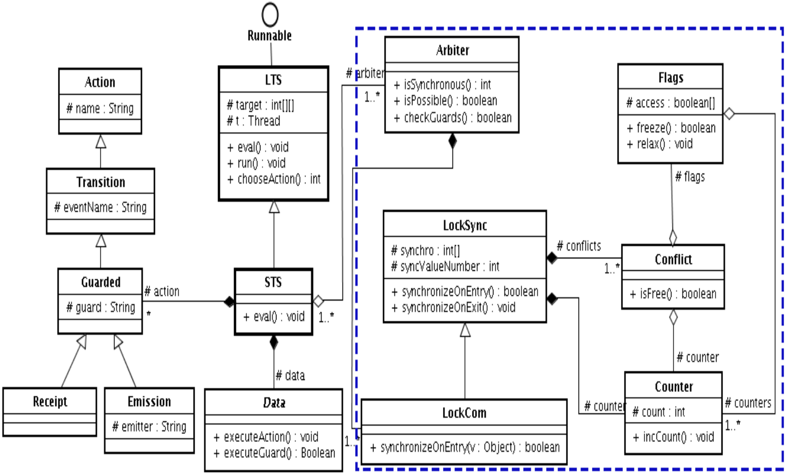
notify()

notify()

data.executeAction("gives",v)

data.executeAction("use",v)

Partial UML Class Diagram



Runnable

LTS
target : int[][]
t : Thread
+ eval() : void
+ run() : void
+ chooseAction() : int

Action
name : String

Transition
eventName : String

Guarded
guard : String

Receipt

Emission
emitter : String

STS
+ eval() : void

Data
+ executeAction() : void
+ executeGuard() : Boolean

Arbiter
+ isSynchronous() : int
+ isPossible() : boolean
+ checkGuards() : boolean

Flags
access : boolean[]
+ freeze() : boolean
+ relax() : void

LockSync
synchro : int[]
syncValueNumber : int
+ synchronizeOnEntry() : boolean
+ synchronizeOnExit() : void

Conflict
+ isFree() : boolean

Counter
count : int
+ incCount() : void

LockCom
+ synchronizeOnEntry(v : Object) : boolean

Related work

- Use of explicit behavioural protocols
    - PROCOL: sequences of events, data types and guards, 1-1 communication
    - SOFA: sequences of events, synchronous communications 1-1 RPC calls
    - Cooperative Objects: Petri-Net, data types and guards, synchronous communications 1-1 RPC calls
- Finite State Processes (FSP) with Java constructions: process algebra based CSP, synchronization based on rendezvous mechanism
- JCSP: provides a CSP model for the Java thread model, Java library, shared channels to synchronize processes, safer alternative than threads

Related work

- Use of explicit behavioural protocols
  - PROCOL: sequences of events, data types and guards, 1-1 communication
  - SOFA: sequences of events, synchronous communications 1-1 RPC calls
  - Cooperative Objects: Petri-Net, data types and guards, synchronous communications 1-1 RPC calls
- Finite State Processes (FSP) with Java constructions: process algebra based CSP, synchronization based on rendezvous mechanism
- JCSP: provides a CSP model for the Java thread model, Java library, shared channels to synchronize processes, safer alternative than threads

Related work

- Use of explicit behavioural protocols
    - PROCOL: sequences of events, data types and guards, 1-1 communication
    - SOFA: sequences of events, synchronous communications 1-1 RPC calls
    - Cooperative Objects: Petri-Net, data types and guards, synchronous communications 1-1 RPC calls
- Finite State Processes (FSP) with Java constructions: process algebra based CSP, synchronization based on rendezvous mechanism
- JCSP: provides a CSP model for the Java thread model, Java library, shared channels to synchronize processes, safer alternative than threads

Related work

- Use of explicit behavioural protocols
  - PROCOL: sequences of events, data types and guards, 1-1 communication
  - SOFA: sequences of events, synchronous communications 1-1 RPC calls
  - Cooperative Objects: Petri-Net, data types and guards, synchronous communications 1-1 RPC calls
- Finite State Processes (FSP) with Java constructions: process algebra based CSP, synchronization based on rendezvous mechanism
- JCSP: provides a CSP model for the Java thread model, Java library, shared channels to synchronize processes, safer alternative than threads

Related work

- Use of explicit behavioural protocols
  - PROCOL: sequences of events, data types and guards, 1-1 communication
  - SOFA: sequences of events, synchronous communications 1-1 RPC calls
  - Cooperative Objects: Petri-Net, data types and guards, synchronous communications 1-1 RPC calls
- Finite State Processes (FSP) with Java constructions: process algebra based CSP, synchronization based on rendezvous mechanism
- JCSP: provides a CSP model for the Java thread model, Java library, shared channels to synchronize processes, safer alternative than threads

Related work

- Use of explicit behavioural protocols
  - PROCOL: sequences of events, data types and guards, 1-1 communication
  - SOFA: sequences of events, synchronous communications 1-1 RPC calls
  - Cooperative Objects: Petri-Net, data types and guards, synchronous communications 1-1 RPC calls
- Finite State Processes (FSP) with Java constructions: process algebra based CSP, synchronization based on rendezvous mechanism
- JCSP: provides a CSP model for the Java thread model, Java library, shared channels to synchronize processes, safer alternative than threads

Conclusions

- Provides an operational interpreter to program primitive components in Java with STS and a powerful way to compose them
- Protocols as Symbolic Transition Systems with full data types, guards and communications (relating verification and execution of component systems)
- Definition of conditional rendezvous taking into account the communicated values
- No constraints on the ordering of processes
- Dynamic checker: to compare generated events to the synchronization rules and compatble with each running state machine
- Efficiency has been partly taken into account: distributing the central arbiter in several objects and minimizing the synchronized parts

Conclusions

- Provides an operational interpreter to program primitive components in Java with STS and a powerful way to compose them
- Protocols as Symbolic Transition Systems with full data types, guards and communications (relating verification and execution of component systems)
- Definition of conditional rendezvous taking into account the communicated values
- No constraints on the ordering of processes
- Dynamic checker: to compare generated events to the synchronization rules and compatble with each running state machine
- Efficiency has been partly taken into account: distributing the central arbiter in several objects and minimizing the synchronized parts

Conclusions

- Provides an operational interpreter to program primitive components in Java with STS and a powerful way to compose them
- Protocols as Symbolic Transition Systems with full data types, guards and communications (relating verification and execution of component systems)
- Definition of conditional rendezvous taking into account the communicated values
- No constraints on the ordering of processes
- Dynamic checker: to compare generated events to the synchronization rules and compatble with each running state machine
- Efficiency has been partly taken into account: distributing the central arbiter in several objects and minimizing the synchronized parts

Conclusions

- Provides an operational interpreter to program primitive components in Java with STS and a powerful way to compose them
- Protocols as Symbolic Transition Systems with full data types, guards and communications (relating verification and execution of component systems)
- Definition of conditional rendezvous taking into account the communicated values
- No constraints on the ordering of processes
- Dynamic checker: to compare generated events to the synchronization rules and compatble with each running state machine
- Efficiency has been partly taken into account: distributing the central arbiter in several objects and minimizing the synchronized parts

Conclusions

- Provides an operational interpreter to program primitive components in Java with STS and a powerful way to compose them
- Protocols as Symbolic Transition Systems with full data types, guards and communications (relating verification and execution of component systems)
- Definition of conditional rendezvous taking into account the communicated values
- No constraints on the ordering of processes
- Dynamic checker: to compare generated events to the synchronization rules and compatble with each running state machine
- Efficiency has been partly taken into account: distributing the central arbiter in several objects and minimizing the synchronized parts

Conclusions

- Provides an operational interpreter to program primitive components in Java with STS and a powerful way to compose them
- Protocols as Symbolic Transition Systems with full data types, guards and communications (relating verification and execution of component systems)
- Definition of conditional rendezvous taking into account the communicated values
- No constraints on the ordering of processes
- Dynamic checker: to compare generated events to the synchronization rules and compatble with each running state machine
- Efficiency has been partly taken into account: distributing the central arbiter in several objects and minimizing the synchronized parts

Future Work

- Definition of a Java based language with STS, asynchronous and synchronous communications
- Current version: reflexivity used to glue protocols and data parts. Compiler version: direct call to the data parts methods
- True usable system: exception handling, barrier optimizations and RMI
- Prove the correctness of the solution
- Use of this new approach into the AMPLE Project

Future Work

- Definition of a Java based language with STS, asynchronous and synchronous communications
- Current version: reflexivity used to glue protocols and data parts. Compiler version: direct call to the data parts methods
- True usable system: exception handling, barrier optimizations and RMI
- Prove the correctness of the solution
- Use of this new approach into the AMPLE Project

Future Work

- Definition of a Java based language with STS, asynchronous and synchronous communications
- Current version: reflexivity used to glue protocols and data parts. Compiler version: direct call to the data parts methods
- True usable system: exception handling, barrier optimizations and RMI
- Prove the correctness of the solution
- Use of this new approach into the AMPLE Project

Future Work

- Definition of a Java based language with STS, asynchronous and synchronous communications
- Current version: reflexivity used to glue protocols and data parts. Compiler version: direct call to the data parts methods
- True usable system: exception handling, barrier optimizations and RMI
- Prove the correctness of the solution
- Use of this new approach into the AMPLE Project

Future Work

- Definition of a Java based language with STS, asynchronous and synchronous communications
- Current version: reflexivity used to glue protocols and data parts. Compiler version: direct call to the data parts methods
- True usable system: exception handling, barrier optimizations and RMI
- Prove the correctness of the solution
- Use of this new approach into the AMPLE Project

Questions?

- Questions?

Components with Symbolic Transition Systems: a Java Implementation of Rendezvous

Fabrício de Alexandria Fernandes
Jean-Claude Royer Robin Passama

École des Mines de Nantes
Department of Computer Science – OBASCO Group
INRIA Research Centre Rennes - Bretagne Atlantique – LINA

10-07-2007 / CPA 2007