Components with Symbolic Transition Systems: a Java Implementation of Rendezvous

Fabréncio 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

Outline

- Introduction
  - Motivation
  - Our work
- STS-Oriented Component Model
  - STS Model
  - Example
- Model Implementation Overview
  - Implementation of the STS
  - Implementation of the Process primitive component
- A Java Implementation of Rendezvous
  - Basic Barrier Principles
  - Synchronization Barrier
  - Evolution of the mechanism development
- Conclusions
Components with STS: a Java Implementation

Fabrício Fernandes, Jean-Claude Royer, Robin Passama

Outline

• Introduction
  • Motivation
  • Our work

• STS-Oriented Component Model
  • STS Model
  • Example

• Model Implementation Overview
  • Implementation of the STS
  • Implementation of the Process primitive component

• A Java Implementation of Rendezvous
  • Basic Barrier Principles
  • Synchronization Barrier
  • Evolution of the mechanism development

• Conclusions
Motivation

- Component Based Software Engineering (CBSE)
- Explicit protocols integrated to 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
Components with STS: a Java Implementation

Fabrício Fernandes, Jean-Claude Royer, Robin Passama

Outline

• Introduction
  • Motivation
  • Our work

• STS-Oriented Component Model
  • STS Model
  • Example

• Model Implementation Overview
  • Implementation of the STS
  • Implementation of the Process primitive component

• A Java Implementation of Rendezvous
  • Basic Barrier Principles
  • Synchronization Barrier
  • Evolution of the mechanism development

• Conclusions
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)

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
Example of STS component

![Diagram of STS component](image-url)
Composition architecture

**p1: process**
- / A:=0:int
- ? think T:int
- / A:=T
- activityOut

**p2: process**
- Same STS as p1

**s: server**
- / S, T, C:=0:int
- ! givet T:int
- / T:=(T+1)%MAXINT
- [C==0]
- ! gives S:int
- / C:=C+1
- end
- / S:=(S+1)%MAXINT
- / C:=C−1

Outline
- Introduction
- STS-oriented Component Model
- Introduction
- Example
- Rendezvous principle
- Model Implementation Overview
- A Java Implementation of Rendezvous
- Conclusions
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
- Guard with receipt: components can conditionally receive and synchronize on a value in the same logical time
Components with STS: a Java Implementation

Fabrício Fernandes, Jean-Claude Royer, Robin Passama

Outline

• Introduction
  • Motivation
  • Our work
• STS-Oriented Component Model
  • STS Model
  • Example
• Model Implementation Overview
  • Implementation of the STS
  • Implementation of the Process primitive component
• A Java Implementation of Rendezvous
  • Basic Barrier Principles
  • Synchronization Barrier
  • Evolution of the mechanism development
• Conclusions
Implementation of the STS

- 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 on the state machine part
- Emitter: pure function computing the emitted value in a given state of the component
- Guard: boolean function implementing a condition
Implementation of the Process Primitive Component

```java
class Process extends Data {
    ...  
}
```

Java Interface

Component Interface

Activity In

Think T:int

Use S:int

Activity Out

End

End
Rules to Generate Interfaces

- Translation rules for one emission and one receipt

  \[
  \text{[guard] event } !\text{emitter:Type / action}
  \right\}
  \begin{align*}
  &\text{public boolean guard();} \\
  &\text{public Type emitter();} \\
  &\text{public void action(}\text{Type var});
  \end{align*}

  \[
  \text{[guard] event } ?\text{var:Type / action}
  \right\}
  \begin{align*}
  &\text{public boolean guard(}\text{Type var}); \\
  &\text{public void action(}\text{Type var});
  \end{align*}

- Automatic generation from STS to Java squeleton

```java
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

```java
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
Components with STS: a Java Implementation

Fabrício Fernandes, Jean-Claude Royer, Robin Passama

Outline

Introduction
- Motivation
- Our work

STS-Oriented Component Model
- STS Model
- Example

Model Implementation Overview
- Implementation of the STS
- Implementation of the Process primitive component

A Java Implementation of Rendezvous
- Basic Barrier Principles
- Synchronization Barrier
- Evolution of the mechanism development

Conclusions
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
- Our solution: one barrier to enter and other one to leave
Basic barrier

LTS
# actions : String []
# target : int [] []
# currentState : int
# thread : Thread
+ void eval()
+ void run()

Arbiter
# counter : int []
# syncValueNumber : int []
+ void synchronizeOnEntry(int action)
  {synchronized}
+ void synchronizeExit(int action)
  {synchronized}
Synchronization barrier

```java
def synchronized public void synchronizeOnEntry(int action) {
    if (counter[action] < syncValueNumber[action] - 1) {
        counter[action]++;
        try {
            wait();
        } catch (InterruptedException e) {
            // we are the last thread
        } catch (InterruptedException e) {
            // so block
        }
    } else {
        counter[action] = 0;
        notifyAll();
        // we are the last thread
        // so wake up all
    }
}
```
Components with STS: a Java Implementation

Fabrício Fernandes, Jean-Claude Royer, Robin Passama

Outline
Introduction
STS-oriented Component Model
Model Implementation Overview
A Java Implementation of Rendezvous

Basic Barrier Principles
Synchronization barrier
Improvements on the mechanism

Conclusions

Sequence Entering the Barrier

```
s: STS
  eval()
  synchronizeOnEntry(): true
  wait()
  synchronizeOnEntry()
  notify()
  notify()
  data.executeAction("gives",v)
  data.executeAction("use",v)
  eval()
p1: STS
  synchronizeOnEntry()
  notify()
  data.executeAction("use",v)
```
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
- 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
Sequence Entering the Barrier

Components with STS: a Java Implementation

Fabrício Fernandes, Jean-Claude Royer, Robin Passama

Outline
Introduction
STS-oriented Component Model
Model Implementation
Overview
A Java Implementation of Rendezvous
Basic Barrier Principles
Synchronization barrier
Improvements on the mechanism
Conclusions
Guards with Communication

- Abstract class Data: execution of guards, emitters and actions on a 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

```
<table>
<thead>
<tr>
<th>s : STS</th>
<th>: Flags</th>
<th>: Arbiter</th>
<th>lc : LockComm</th>
</tr>
</thead>
<tbody>
<tr>
<td>lc : LockComm = isSynchronous(&quot;gives&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data.executeGuard(&quot;gives&quot;) : true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v = computeEmittedValue(&quot;gives&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>setEmittedValue(v)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronizeOnEntry() : true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isPossible(lc) : true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkGuards(lc) : true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>freeze() : true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>relax()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wait()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lc : LockComm = isSynchronous(&quot;use&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v = getEmittedValue(&quot;use&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>synchronizeOnEntry()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>conflict.isFree() : true</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>notify()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>notify()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data.executeAction(&quot;gives&quot;, v)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data.executeAction(&quot;use&quot;, v)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Partial UML Class Diagram
Components with STS: a Java Implementation

Fabrício Fernandes, Jean-Claude Royer, Robin Passama

Outline

Introduction
  • Motivation
  • Our work
STS-Oriented Component Model
  • STS Model
  • Example
Model Implementation Overview
  • Implementation of the STS
  • Implementation of the Process primitive component
A Java Implementation of Rendezvous
  • Basic Barrier Principles
  • Synchronization Barrier
  • Evolution of the mechanism development
Conclusions

Final remarks
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
• Solution providing a mechanism to synchronize component with protocols
• 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
Future Work

- Definition of a Java based language with STS, asynchronous and synchronous communications
- True usable system: exception handling, barrier optimizations and RMI
- Prove the correctness of the solution
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