



# Scenario Modeling with Aspects

João Araújo

CITI/ Departamento de Informática  
Universidade Nova de Lisboa (UNL), Portugal  
ja@di.fct.unl.pt

## Summary

### ■ 1st Part: Early Aspects – Main Concepts

- ◆ In collaboration with
  - Ana Moreira (UNL)
  - Awais Rashid (Lancaster University)
  - Bedir Tekinerdogan (University of Twente)
  - Elisa Baniassad (Chinese University of Hong Kong)
  - Paul Clements (SEI)

### ■ 2nd Part: Scenario Modelling with Aspects

- ◆ In collaboration with
  - Jon Whittle (QSS/NASA Ames, San Francisco)
  - Dae-Kyoo Kim (Oakland University, Michigan)

# AOSD

- The problem of crosscutting concerns (***a concern that cuts across modules***) :
  - ◆ Scattered across several modules
  - ◆ Leads to tangled systems
- AOSD solves this problem by introducing the concept of an *aspect* plus programming-level support to
  - ◆ (a) implement aspects separately from the modules, isolating them in one place where they may be easily managed and changed, and
  - ◆ (b) once defined, to **weave** them back into the modules as though all of the modules each contained a copy of the implementation of the aspect

# Early Aspects

- Work in aspects has been fairly limited to the implementation phase of software development
- Recent work has tried to generalize the concept and apply it to different phases of the life cycle
- In particular, early aspects are crosscutting concerns that are identified in the early life cycle phases
- Nominally these phases include the **domain analysis, requirements elicitation and analysis, and architecture design phases**
- Our focus here is on the early requirements stages

## How a concern crosscuts other concerns at requirements?

- Through scattering and tangling
  - ◆ Scattering in requirements would refer to scattering of the *description* of a concern throughout a requirements set
  - ◆ Tangling at the requirements level manifests as the *description* of a concern tangled with the *description* of another concern

## Early Aspects

- Scattering and tangling reduces the modularity of the artifacts in the early life cycle
- This might lead to serious maintenance problems, or even the failure of the system to meet certain quality attributes
- Since the early SW development phases actually set the early design decisions and have a large impact on the system
  - ◆ Crosscutting concerns must be localized and managed as distinct concepts using Aspect-Oriented Requirements Engineering (AORE) techniques.
  - ◆ Coping with aspects at the early life cycle phases as such is a primary issue

## AORE and Composition

- AORE techniques build upon the strong focus on *composability* in AOSD by providing a fine-grained specification of how a requirements-level aspect constrains or influences specific requirements in a system
- The composition relationships between aspectual and non-aspectual requirements leads to an improved understanding of their interaction, inter-relationships and conflicts
- This helps to identify trade-offs early on in the development life cycle and undertake negotiations with the affected stakeholders



## 2<sup>nd</sup> Part: Scenario Modeling with Aspects

In collaboration with Jon Whittle (QSS/NASA Ames) and  
Dae-Kyoo Kim (Oakland University, Michigan)

# Introduction

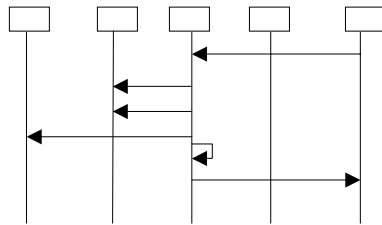
- AOSD: handle crosscutting concerns by:
  - ◆ Systematic identification
  - ◆ Modularisation
  - ◆ Composition
- We address:
  - ◆ Crosscutting requirements scenarios
- How?
  - ◆ **Separate**
  - ◆ **Represent** → interaction pattern specifications (IPs)
  - ◆ **Compose** → customizable composition algorithm
  - ◆ **Simulate** → transform to executable form

# Motivation

- We consider aspects at the requirements level. In particular, we concentrate on scenario-based requirements.
- Scenarios are commonly used in RE because they are easily understood by stakeholders. A scenario is an example trace of desired or existing system behavior.
- A complete and consistent set of scenarios can be difficult to specify, (e.g., exception, failure cases). Many of these are aspectual in the sense that they crosscut other scenarios.
- Our process provides a way to describe aspectual and non-aspectual scenarios independently and then to merge them together for the purpose of validating the complete set of scenarios.

# What is a Scenario?

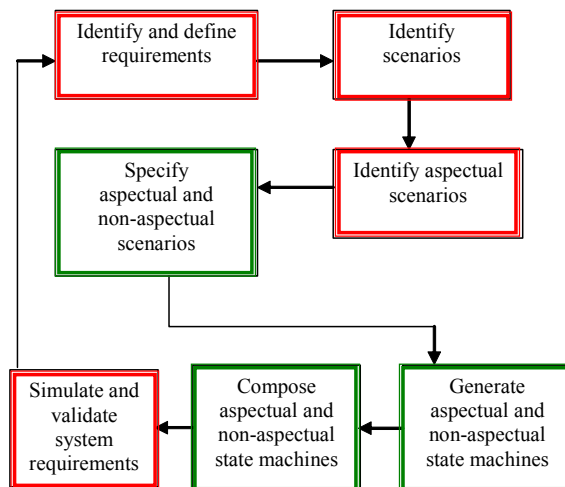
Scenario = example trace of desired or existing behavior



UML Sequence Diagram

- shows the *global* interactions between classes/components
- describe concrete interactions → good for early development and communication
- part of popular software development methodologies (e.g., use-case based)

# Overview of our approach



---

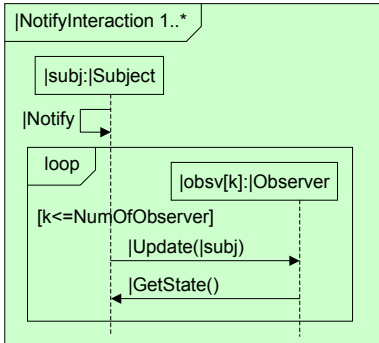
“Represent”

## **Pattern Specifications**

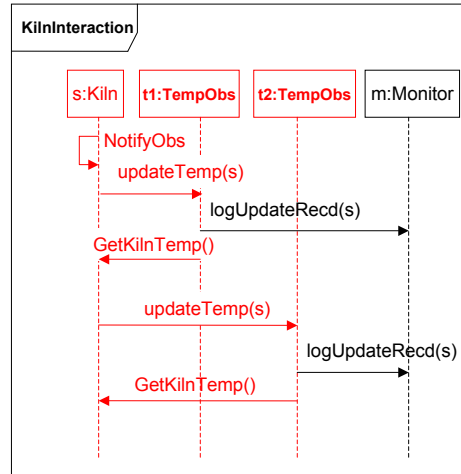
---

- A Pattern Specification (PS) describes a pattern of structure or behavior and is defined in terms of roles
- A role specifies properties that a UML model element must have if it is to be part of the UML model
- A PS can be instantiated by assigning UML model elements to the roles in the PS
- A model conforms to a PS if its model elements that play the roles of the PS satisfy the properties defined by the roles

# Conforming Sequence Diagram



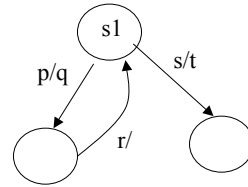
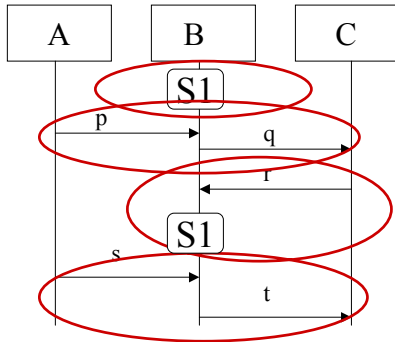
IPS



Conforming SD

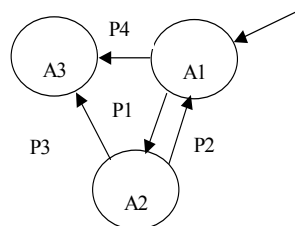
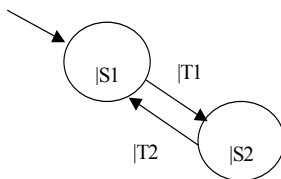
“Compose”

# State Machine Synthesis

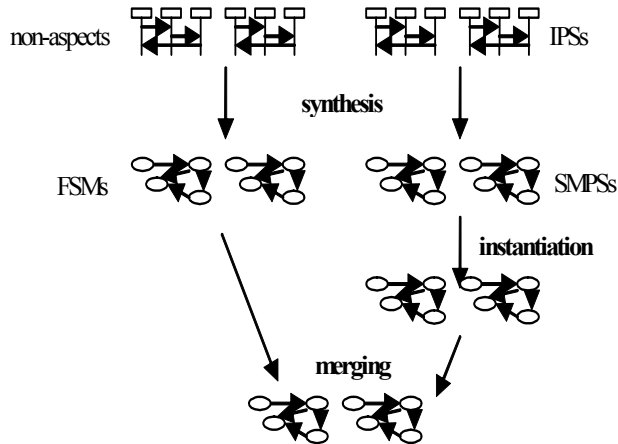


# State Machine Pattern Specifications

- An SMPS defines a pattern of state-based behavior between its participants. It consists of state roles and transition roles
- An SMPS can be instantiated by assigning concrete modeling elements to the roles:
  - ◆ Bind |S1 to A1
  - ◆ Bind |S2 to A2
  - ◆ Bind |T1 to P1
  - ◆ Bind |T2 to P2



## Compose aspectual and non-aspectual FSM

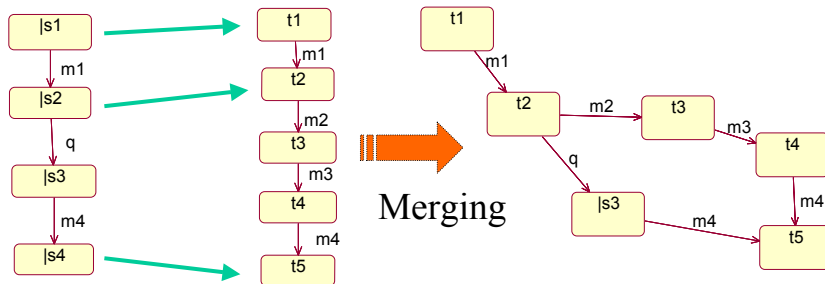


## Instantiation and Merging

- Method for instantiation and merging at the state machine level (i.e., after synthesis takes place)
- Instantiation is a manual process because the bindings have to be provided for each case in which an aspect crosscuts a non-aspect
- Merging, however, can be partially automated
- Our approach is to define an algorithm for automating the process of merging but to allow fine-tuning of this algorithm by user input

# Instantiation/Merging

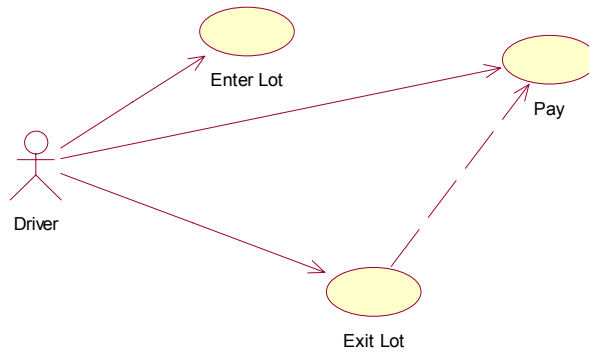
Instantiation:  $\theta(|s1)=t1$ ,  $\theta(|s2) = t2$ ,  $\theta(|s4) = t5$



# The Car Parking System

- A client has to get a ticket from a machine after pressing a button
- Afterwards, the car is allowed to enter and park in an available place
- The system has to control if the car parking is full or if it still has places left
- To leave the parking place, s/he has to pay the ticket obtained in a paying machine
- The amount depends on the time spent. After paying the client can leave by inserting the ticket in a machine which will open the gate
- Regular users of the parking system may pre-purchase time and enter/exit by inserting a card and PIN number which will result in money being deducted automatically from the user's account

# Use Case Diagram for the Car Parking System



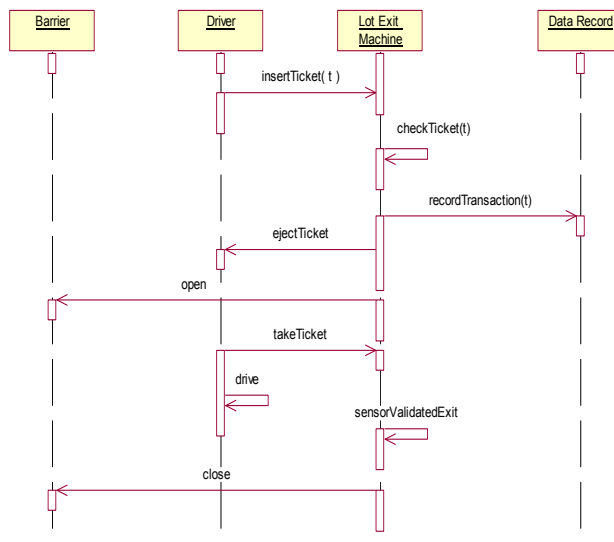
## Non-Aspectual Scenarios

I1	Enter Lot, parking lot has space
I2	Enter Lot, parking lot has no space
I3	Enter Lot, regular user types in PIN and enters
I4	Exit Lot, driver inserts ticket; ticket paid
I5	Exit Lot, driver inserts ticket; ticket not paid
I6	Exit Lot, driver has no ticket
I7	Exit Lot, grace period from paying ticket exceeded
I8	Exit Lot, regular user types in PIN and exits
I9	Exit Lot, driver types in PIN but insufficient funds in account
I10	Pay, driver inserts ticket and correct money
I11	Pay, driver adds money to PIN card

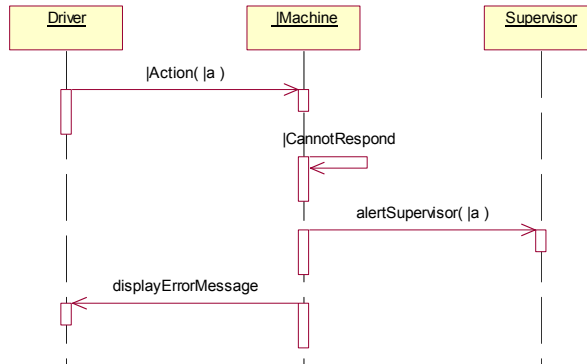
# Aspectual Scenarios

A1	Machine is broken
A2	Ticket cannot be read
A3	PIN incorrect

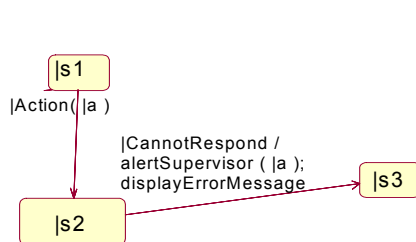
# Sequence Diagram for exiting with paid ticket



# IPS for “Machine is broken”

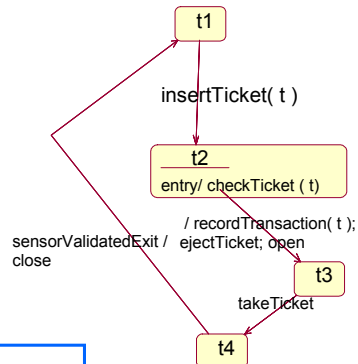


# Generating SMPs and FSMs



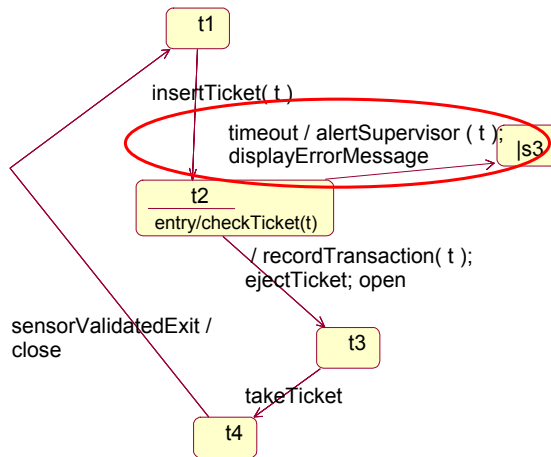
SMPs for |Machine

1. |s1 binds to t1
2. |s2 binds to t2
3. |Action binds to insertTicket
4. |a binds to t
5. |CannotRespond binds to timeout

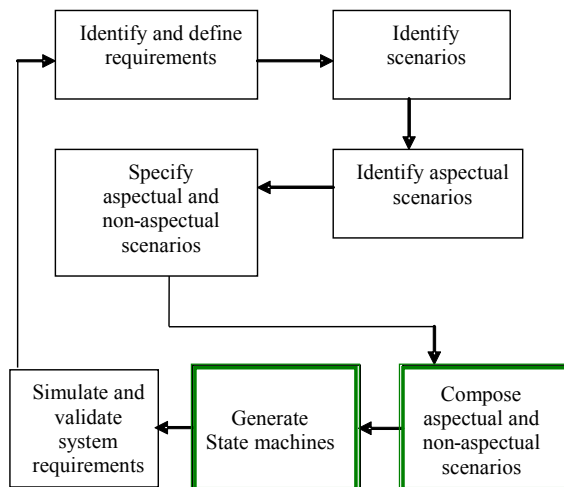


State Machine for Lot Exit Machine

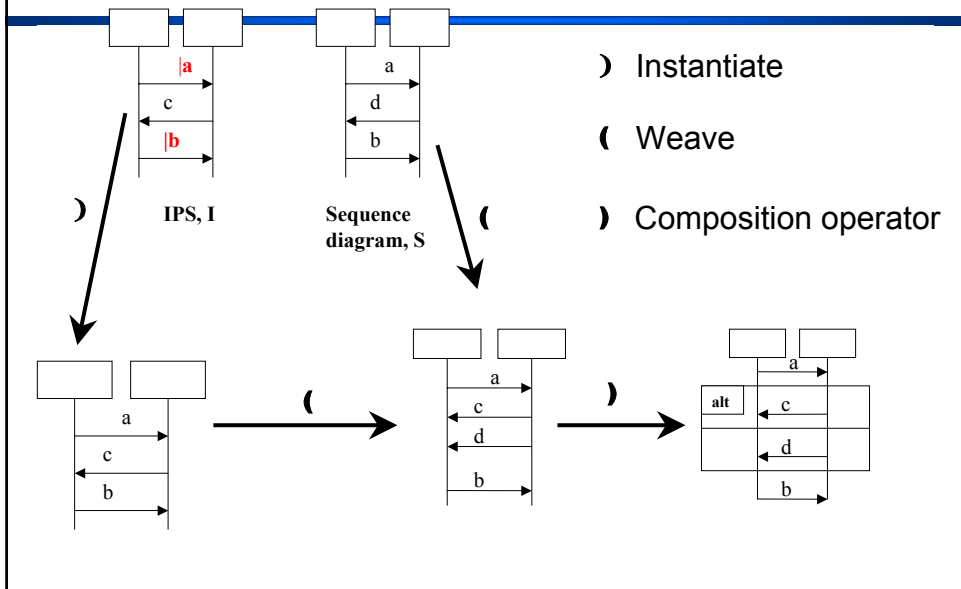
# Composing Aspectual and Non-Aspectual FSMs



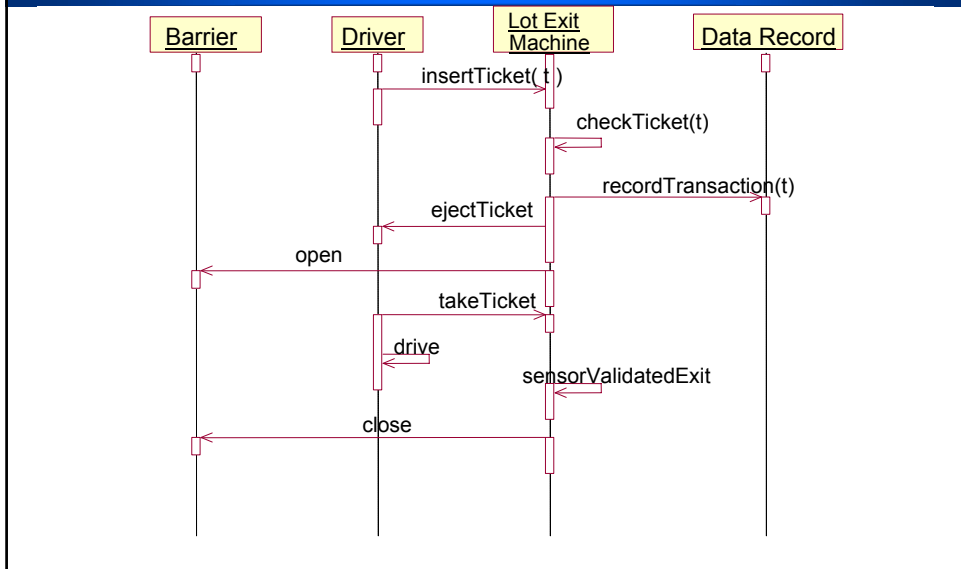
## Alternative approach



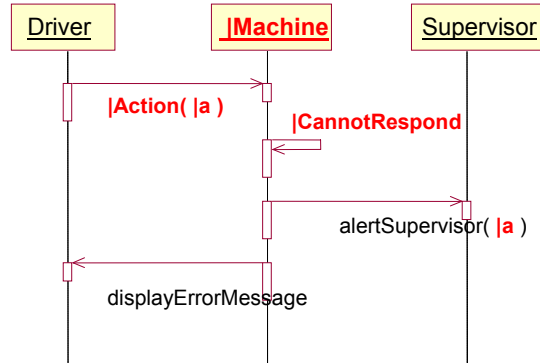
# Composition at interaction level



# Car parking example

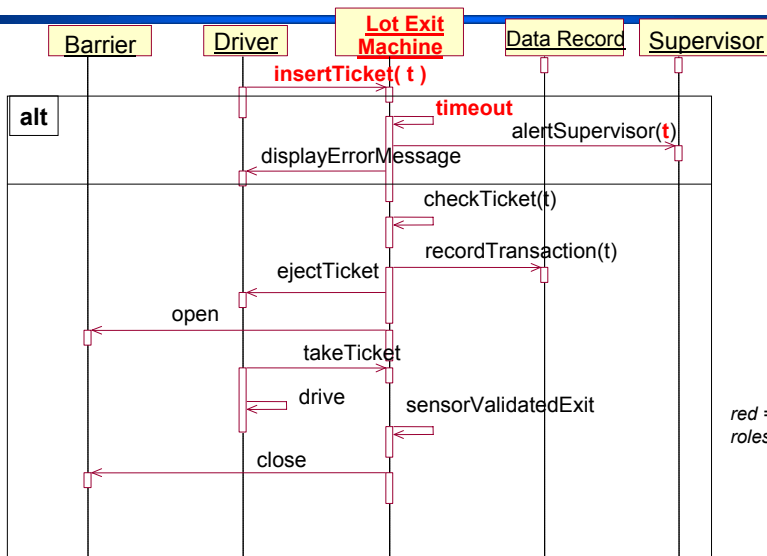


# Instantiation



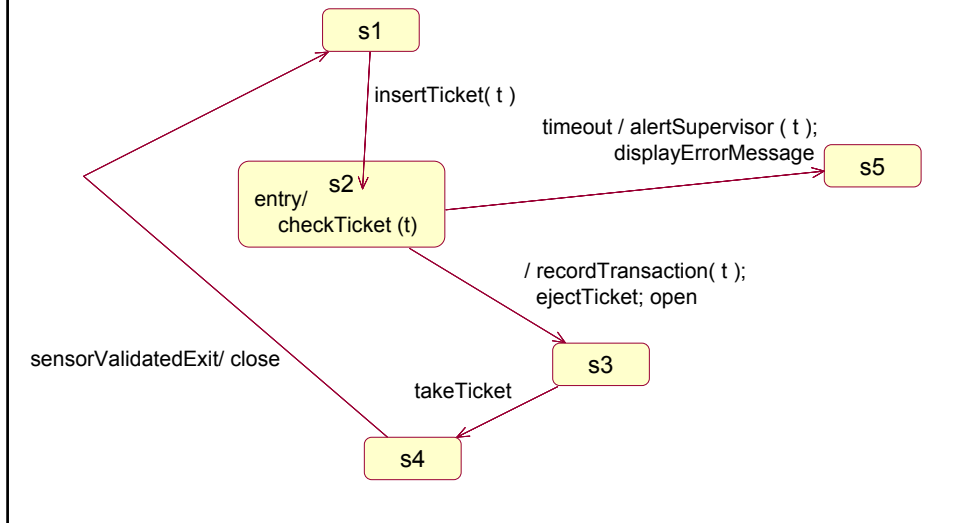
|Machine binds to Lot Exit Machine  
 |Action binds to insertTicket  
 |a binds to t  
 |CannotRespond binds to timeout

# Composed Interaction



red = former roles

## State machine generated



## Composition Operators

- **OR**: alternative interactions with choice point to decide
- **AND**: interactions occur concurrently
- **IN**: insert one interaction inside the other
- Others: further work...

## Conclusions

- We presented an approach to modeling scenario-based requirements using aspect-oriented principles
- Aspectual scenarios were modeled using Interaction Pattern Specifications (IPSS).
- A technique was described to compose aspectual and non-aspectual scenarios and to transform them into a set of executable state machines.
- In this way, we showed how to separate aspects during scenario development but also how to generate a composed behavioral description for simulating the scenarios

## Future work

- For future work we will investigate
  - ◆ How to use the result of the simulation step to augment or correct the scenario models
  - ◆ One issue that has not been directly addressed is scalability.
    - The developer must provide binding statements for each aspect and for each scenario that the aspect crosscuts
    - We expect there to be ways to manage this complexity, for example, by providing default bindings
  - ◆ Apply the technique to other UML models



More information at:

[\*http://early-aspects.net/\*](http://early-aspects.net/)

[\*http://ctp.di.fct.unl.pt/AOSD-Group/index.html\*](http://ctp.di.fct.unl.pt/AOSD-Group/index.html)