Modular Design of Man-Machine Interfaces with Larissa

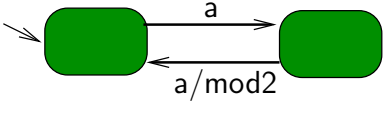
Karine Altisen, Florence Maraninchi, David Stauch
Verimag, Grenoble, France

Introduction

- Reactive systems are systems which are in constant interaction with their environment
- Cross-cutting concerns exist in reactive systems
- Aspect-oriented programming modularizes cross-cutting concerns, but existing aspect languages cannot be used
- Larissa is an aspect language for the synchronous programming language Argos
- This talk : model the interface of a complex wristwatch with Argos and Larissa

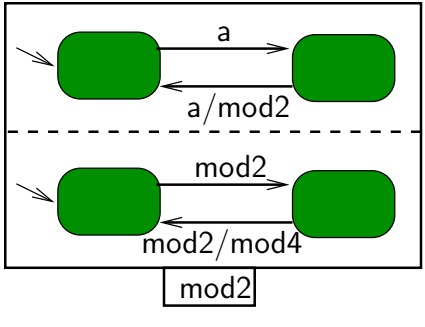
Argos

- **Hierarchical, synchronous automata language**
- **Basic element : complete and deterministic Mealy automata**
- **Interface : a set of inputs, a set of outputs**



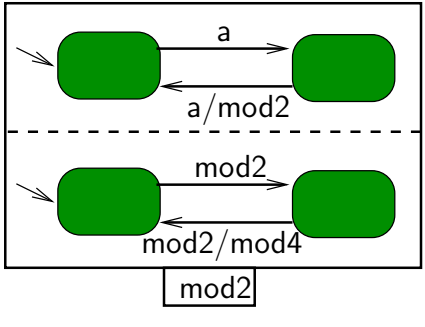
Argos

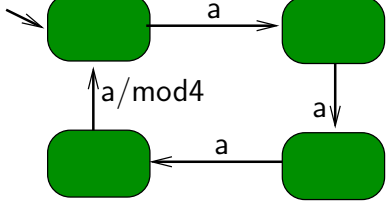
- **Hierarchical, synchronous automata language**
- **Basic element : complete and deterministic Mealy automata**
- **Interface : a set of inputs, a set of outputs**
- **Operators : parallel product, encapsulation, (refinement)**



Argos

- **Hierarchical, synchronous automata language**
- **Basic element : complete and deterministic Mealy automata**
- **Interface : a set of inputs, a set of outputs**
- **Operators : parallel product, encapsulation, (refinement)**
- **Compiled into flat automata**



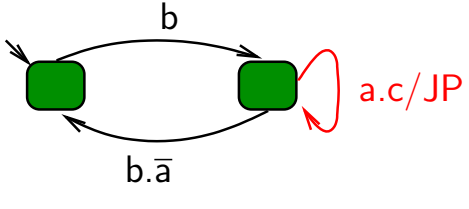


Larissa

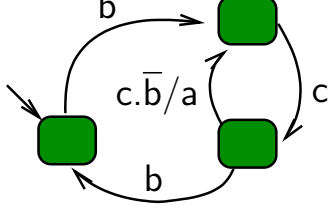
- Aspect language for Argos
- Consists of pointcuts and advice :
  - a join point is a transition
  - pointcuts select transitions in automata
  - advice replaces these transitions
- This cannot be done with the existing operators
- We want to preserve semantic properties, e.g. preservation of equivalence

Pointcuts

- Observer automata which take as inputs the inputs and outputs of the program
- Output JP is emitted when the program is in a join point
- Independent of the implementation of the program



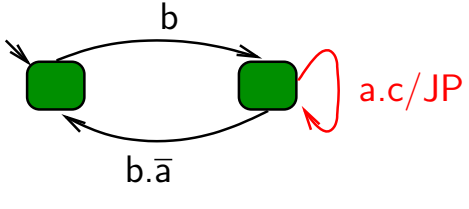
pointcut



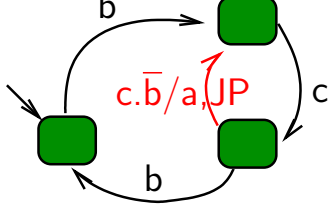
base program

Pointcuts

- Observer automata which take as inputs the inputs and outputs of the program
- Output JP is emitted when the program is in a join point
- Independent of the implementation of the program



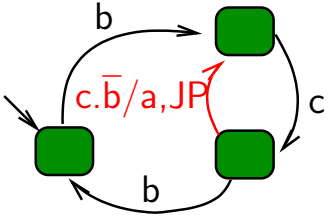
pointcut



join point program

toInit Advice

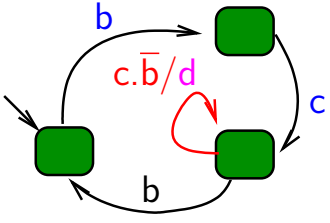
- When a join point is passed, program execution is changed :
  - emit some outputs O
  - go to some target state
  - target state defined by a finite input trace
- Example advice : trace b.c, advice output d



join point program

toInit Advice

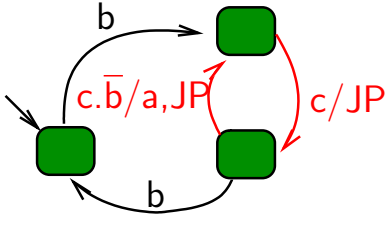
- When a join point is passed, program execution is changed :
  - emit some outputs O
  - go to some target state
  - target state defined by a finite input trace
- Example advice : trace b.c, advice output d



woven program

toCurrent Advice

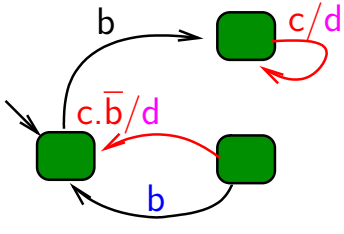
- As toInit, but execute the trace from the source state of the join point
- Example advice : trace b, advice output d



join point program

toCurrent Advice

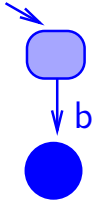
- As toInit, but execute the trace from the source state of the join point
- Example advice : trace **b**, advice output **d**



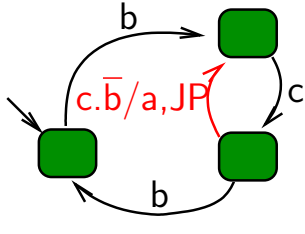
woven program

Advice Program

- **Add an automaton to the join point transition**
- **Example : tolnit advice, trace b.c, output d, inserted automaton**



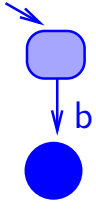
inserted automaton

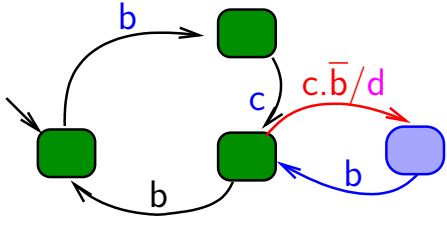


join point program

Advice Program

- **Add an automaton to the join point transition**
- **Example : tolnit advice, trace b.c, output d, inserted automaton**





inserted automaton **woven program**

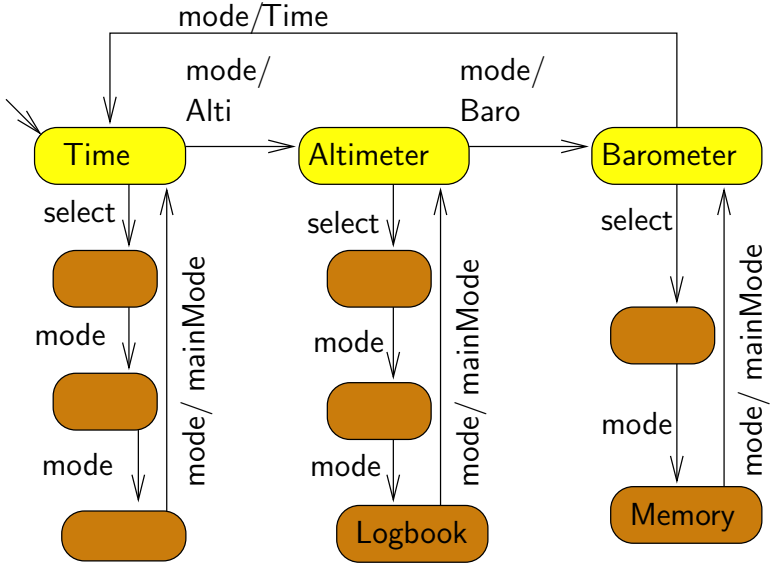
Case Study : Suunto Wristwatches

- **Model the interface of two complex wristwatches : Altimax and Vector**
- **Altimax contains a watch, altimeter, and barometer, the Vector also a compass**
- **Each functionality has a main mode and some submodes**
- **Four buttons :** `mode`, `select`, `minus`, `plus`

Modularization with Aspects

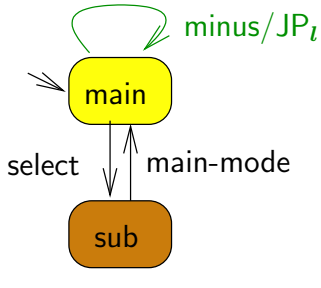
- **We use aspects to build a product line**
- **Models of the product line are build from the base program and aspects**
- **Altimax : base-program ◁alti-shortcut**
- **Vector : base-program ◁compass-mode ◁compass-shortcut**

Base Program

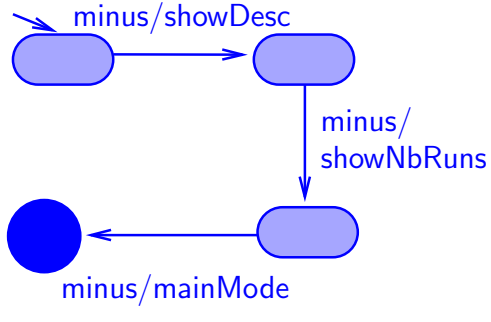


Altimax : alti-shortcut

- `minus` **button is used as a shortcut in the main modes**
- **Pressing** `minus` **goes to the Logbook mode**
- **toCurrent aspect alti-shortcut with empty trace**
- **Output Logbook**
- **inserted automaton shows desired information**



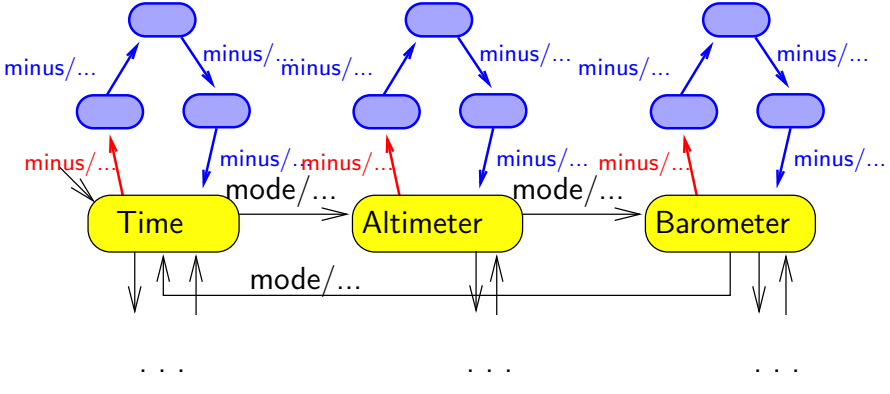
pointcut



inserted automaton

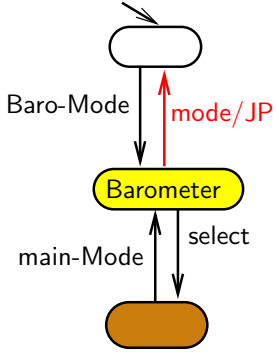
Altimax : Woven Program

– **The weaving inserts an inserted automaton in every main mode**

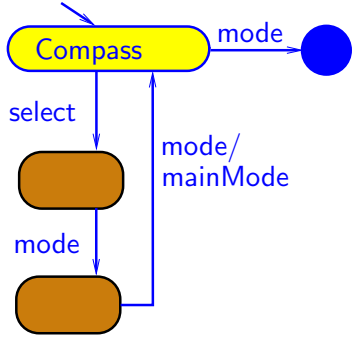


Vector : compass-mode

– **Vector has an additional Compass mode**
– **Use an aspect to add it to the base program**



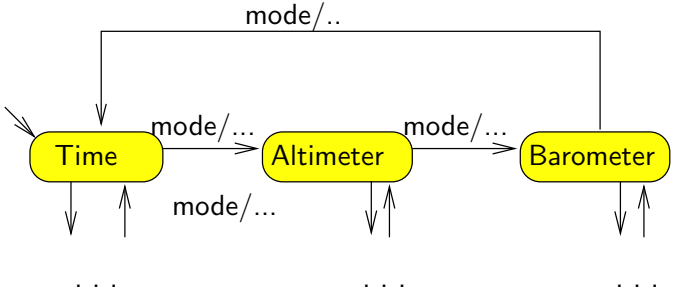
pointcut



inserted automaton

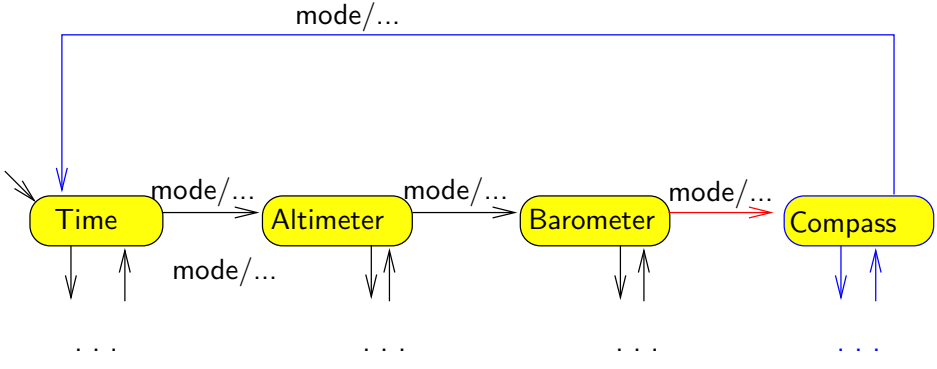
Vector : Woven Program

– **The compass mode aspect adds the compass mode**



Vector : Woven Program

– The compass mode aspect adds the compass mode

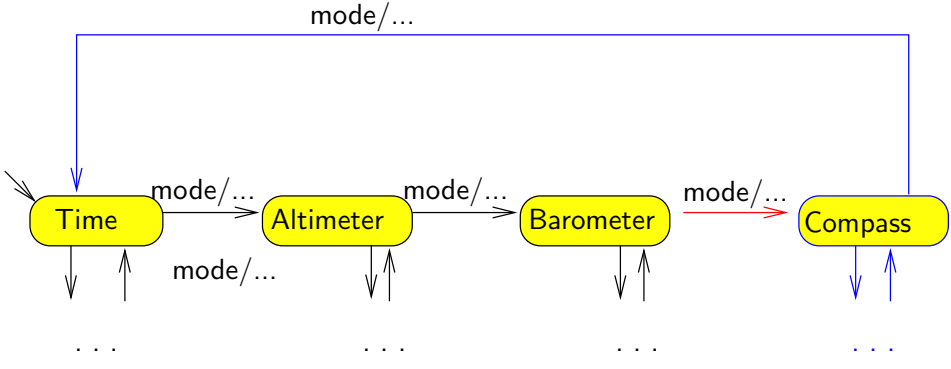


Vector : compass-shortcut

- **Shortcut of the Vector is different from the Altimax shortcut**
- **When the `minus` button is pressed in a main mode, the Vector goes directly to the Compass mode**
- **implemented with a toInit aspect**
  - **same pointcut as the alti-shortcut**
  - **trace `mode.mode.mode.mode`**
  - **output Compass**

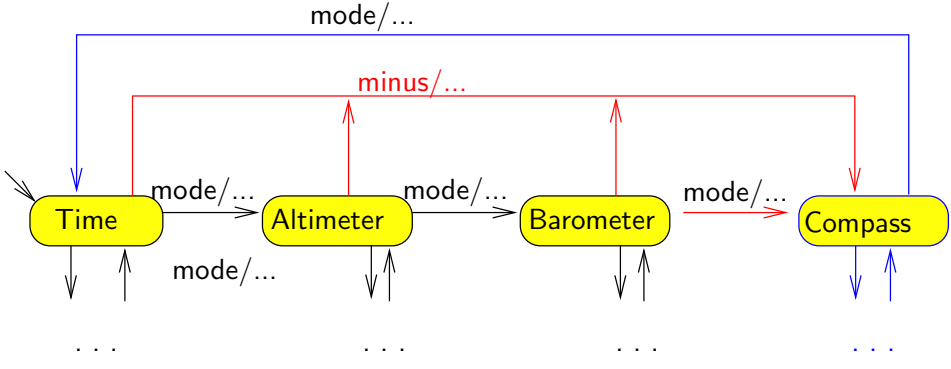
Vector : Woven Program

– The compass mode aspect adds the compass mode



Vector : Woven Program

- **The compass mode aspect adds the compass mode**
- **The compass shortcut aspect adds shortcut transitions to the compass mode**



Modularization with Aspects

- **Altimax : base-program ◁alti-shortcut**
- **Vector : base-program ◁compass-mode ◁compass-shortcut**
- **Aspects encapsulate shortcut concern**
- **avoids code duplication, shortcut code is only written once**
- **shortcuts are modularized, replacing one by the other is easy**

Aspects as Components

- **Claim : consider aspects as normal components**
- **Larissa seems a good candidate :**
  - **aspects can be mixed freely with other components**
  - **aspects refer only to the interface of the programs they advise, thus respecting its encapsulation**
  - **aspects have the same semantic properties as other composition operators**

Conclusion and Perspectives

- Larissa modularizes cross-cutting concerns in the case study
- Larissa seems well suited for feature-oriented development
- A good base for the definition of an semantic aspect component
- Contracts for base components can be specified with observer automata
- An aspect could be specified as a transformation of the contract of the base program into the contract of the woven program