Larissa, an Aspect-Oriented Language for Reactive Systems

PhD Defense

David Stauch

Verimag/Grenoble INP

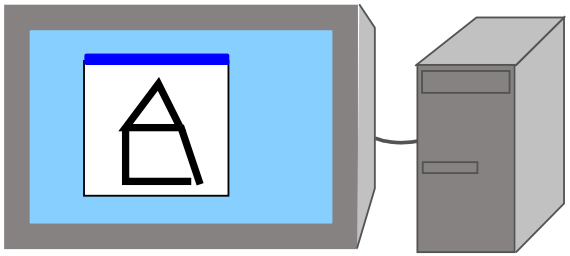
November 13th, 2007

Jury

Roland Groz
Shmuel Katz
Mario Südholt
Pascal Fradet
Florence Maraninchi
Karine Altisen

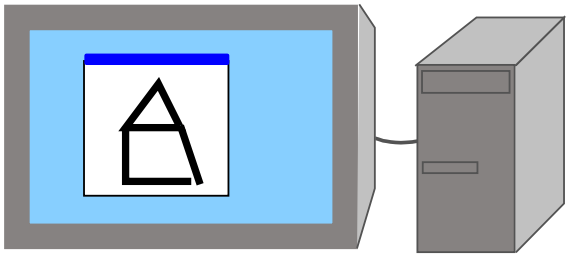
Outline

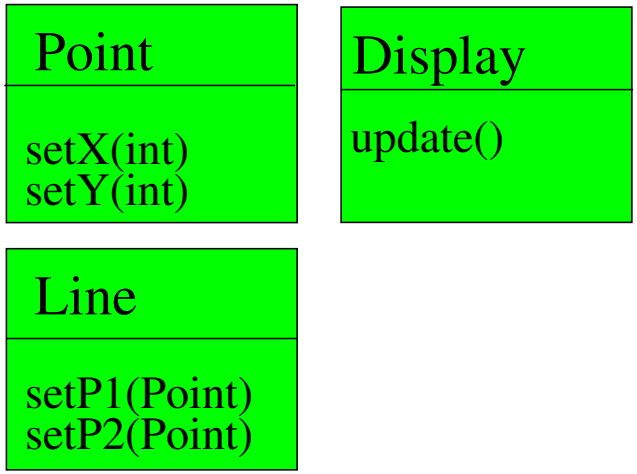
Aspect-Oriented Programming



- Example: figure editor

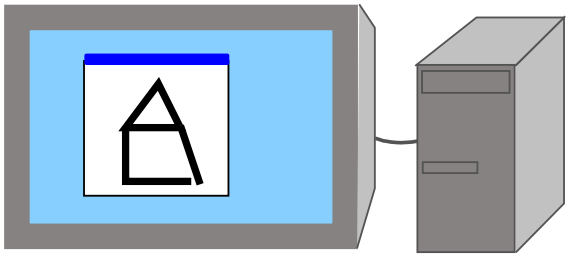
Aspect-Oriented Programming

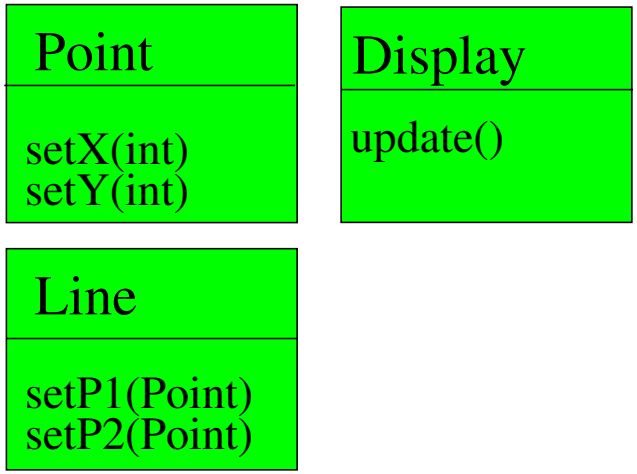




- Example: figure editor
- Program: modules implement concerns

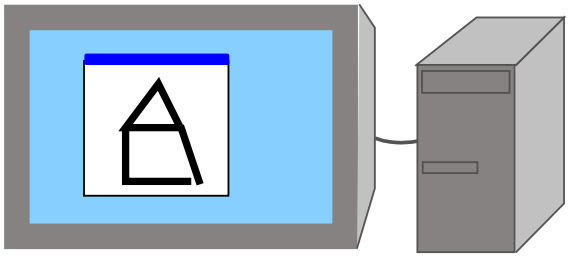
Aspect-Oriented Programming

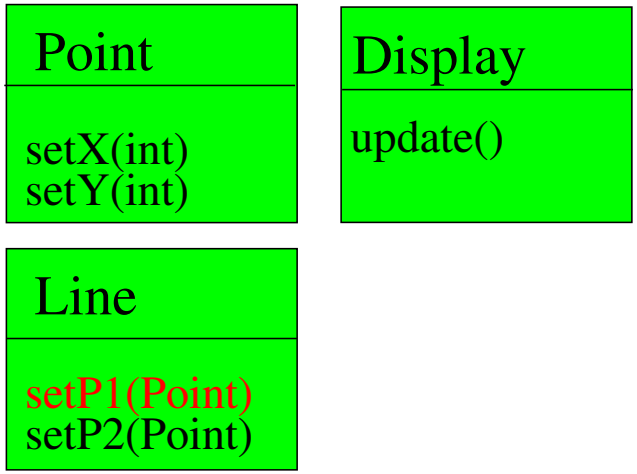




- Example: figure editor
- Program: modules implement concerns
- Crosscutting concerns: cannot be put in own module
- Example: update display

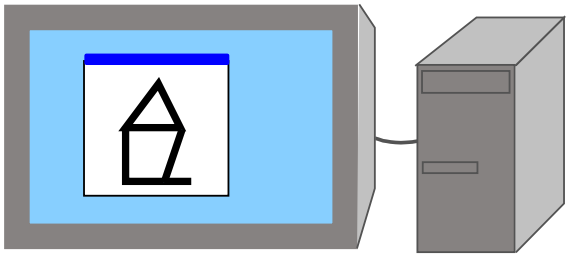
Aspect-Oriented Programming

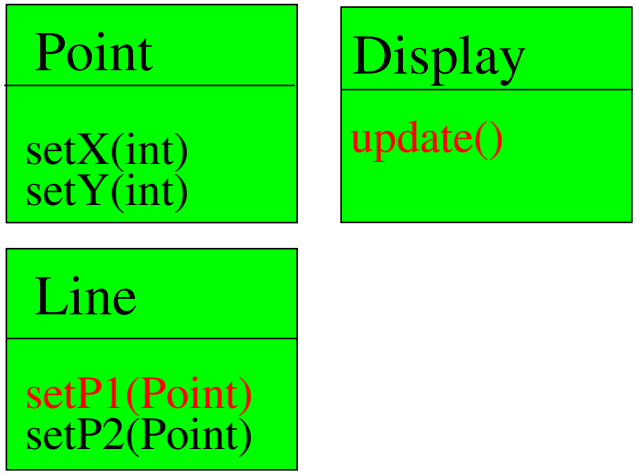




- Example: figure editor
- Program: modules implement concerns
- Crosscutting concerns: cannot be put in own module
- Example: update display

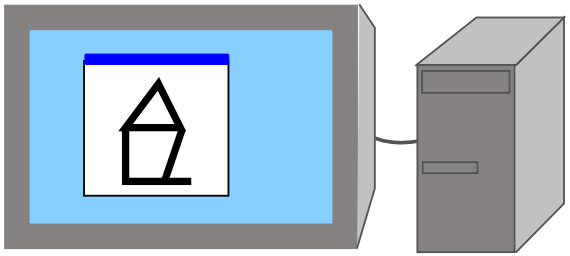
Aspect-Oriented Programming





- Example: figure editor
- Program: modules implement concerns
- Crosscutting concerns: cannot be put in own module
- Example: update display

Aspect-Oriented Programming



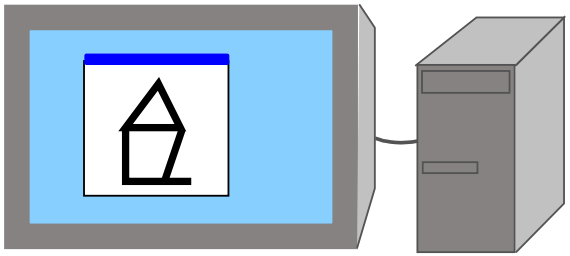
Point
setX(int) setY(int)

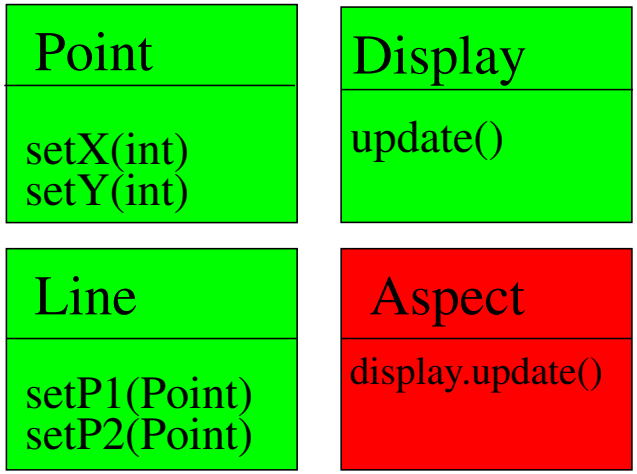
Display
update()

Line
setP1(Point) setP2(Point)

- Example: figure editor
- Program: modules implement concerns
- Crosscutting concerns: cannot be put in own module
- Example: update display
- Solution: Add **update()** to methods
- Bad: scattered code

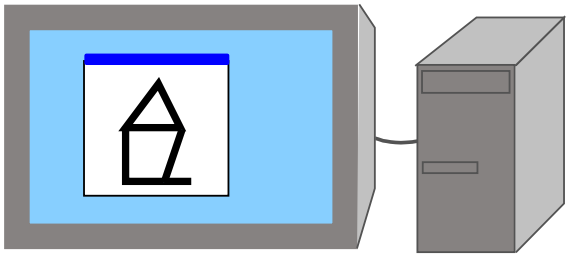
Aspect-Oriented Programming

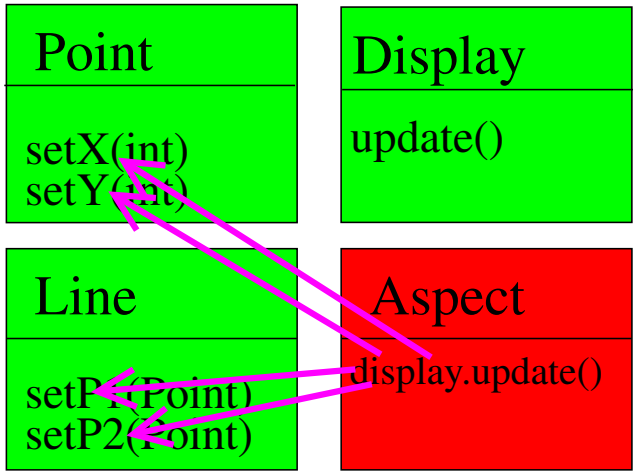




- Example: figure editor
- Program: modules implement concerns
- Crosscutting concerns: cannot be put in own module
- Example: update display
- Solution: Add **update()** to methods
- Bad: scattered code
- Idea: put crosscutting code in aspect

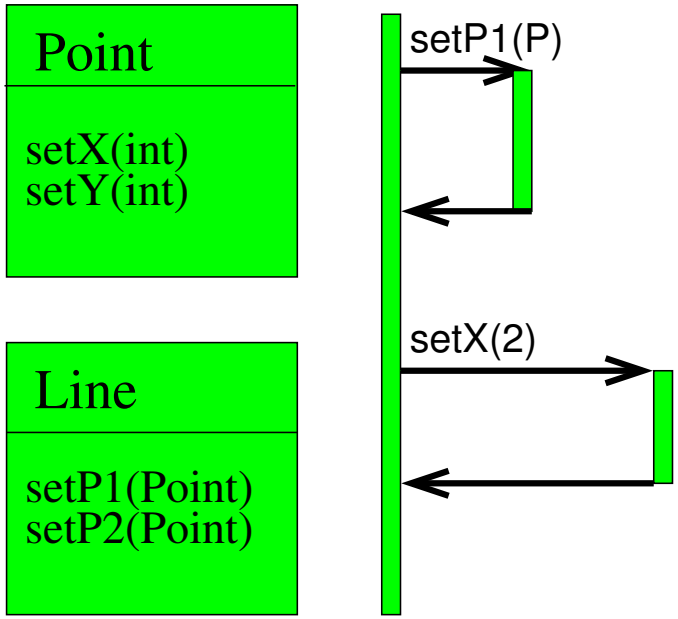
Aspect-Oriented Programming



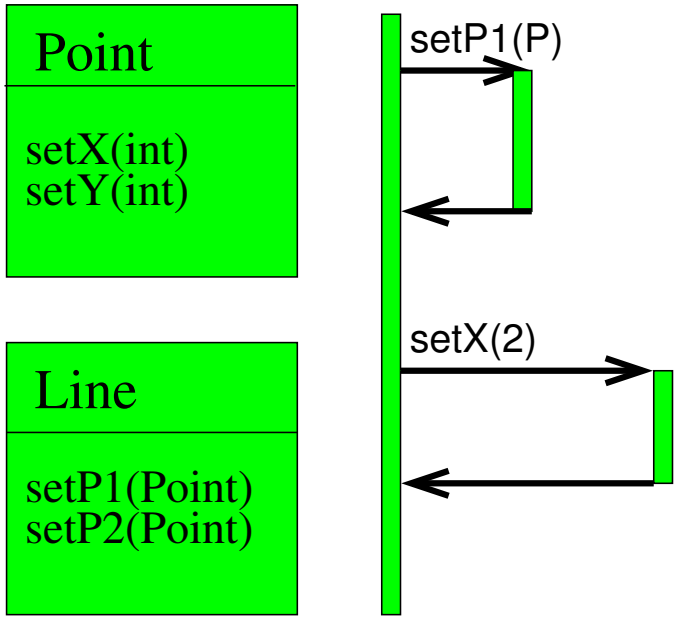


- Example: figure editor
- Program: modules implement concerns
- Crosscutting concerns: cannot be put in own module
- Example: update display
- Solution: Add **update()** to methods
- Bad: scattered code
- Idea: put crosscutting code in aspect
- Weave aspect in program

Example in Java and AspectJ



Example in Java and AspectJ

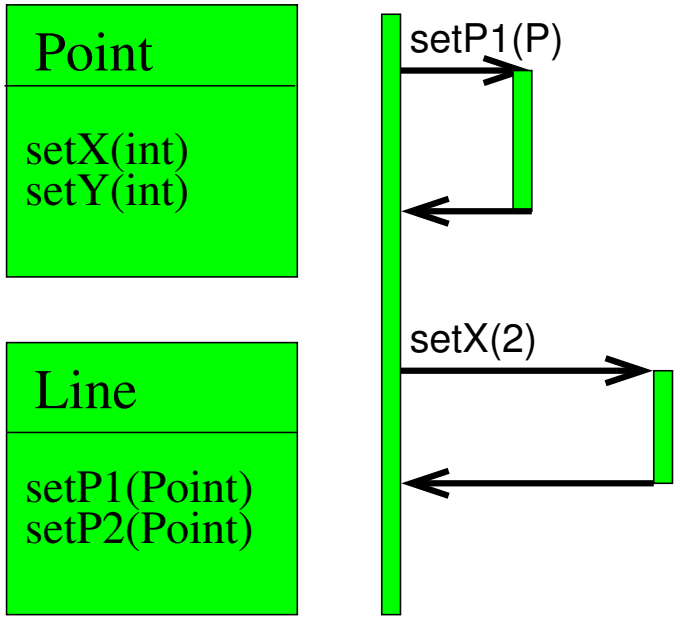



```
aspect updateDisplay{
}
```

AOP: Key Concepts

- Join points: where aspects intervene
- Pointcut: select join points
- Advice: what aspect does

Example in Java and AspectJ

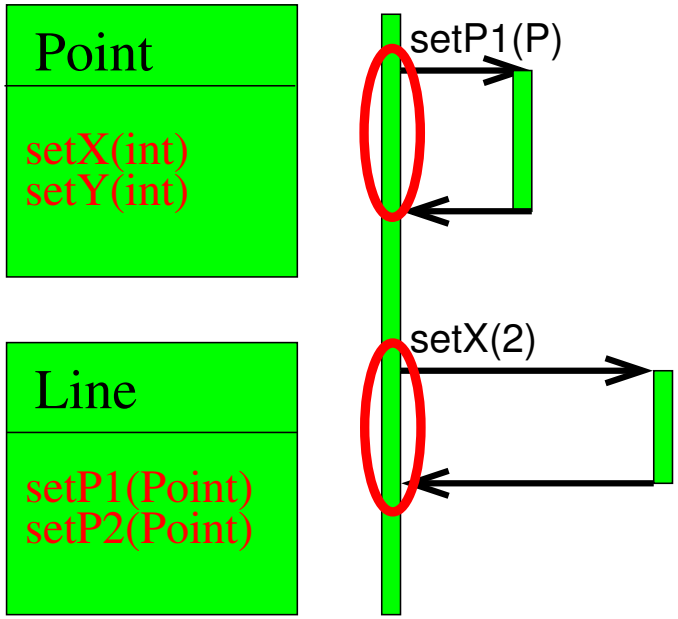



```
aspect updateDisplay{

  pointcut stateChanged() :
    || call(void Point.set*(..))
    || call(void Line.set*(..));

}
```

AOP: Key Concepts

- Join points: where aspects intervene
- Pointcut: select join points
- Advice: what aspect does

Example in Java and AspectJ

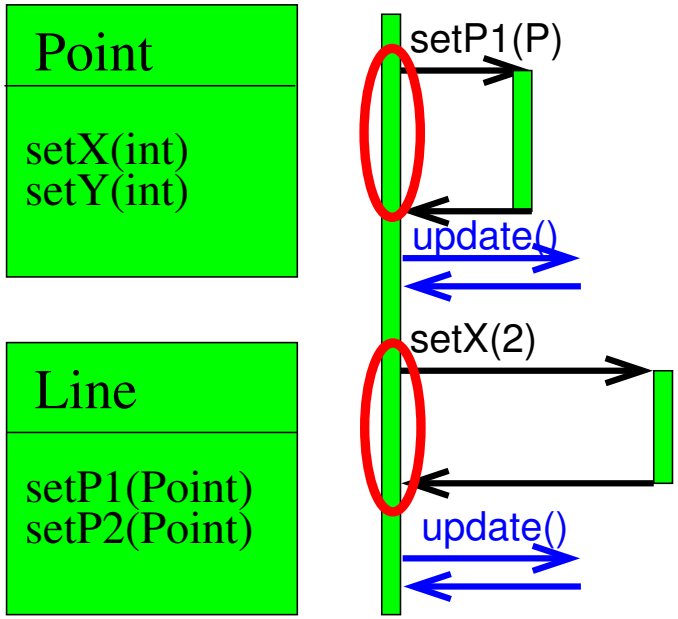



```
aspect updateDisplay{

  pointcut stateChanged() :
    || call(void Point.set*(..))
    || call(void Line.set*(..));

}
```

AOP: Key Concepts

- Join points: where aspects intervene
- Pointcut: select join points
- Advice: what aspect does

Example in Java and AspectJ

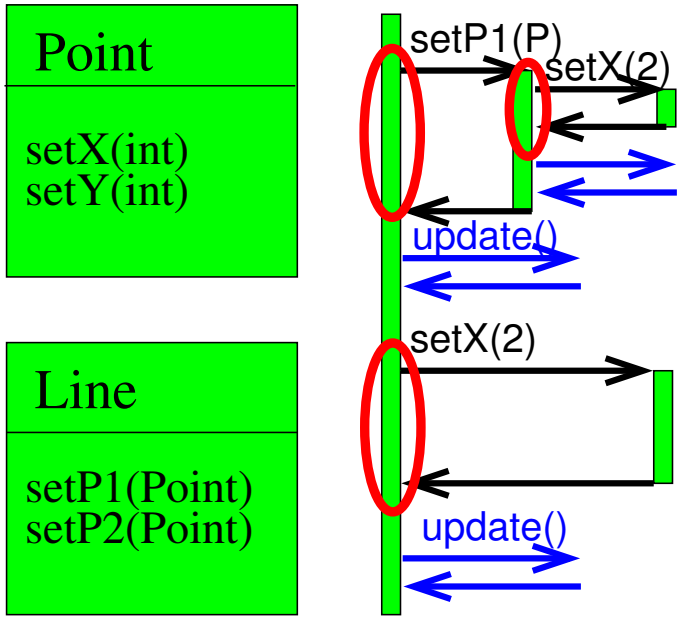



```
aspect updateDisplay{

  pointcut stateChanged() :
    || call(void Point.set*(..))
    || call(void Line.set*(..));

  after() : stateChanged() {

    display.update();
  }
}
```

AOP: Key Concepts

- Join points: where aspects intervene
- Pointcut: select join points
- Advice: what aspect does

Example in Java and AspectJ

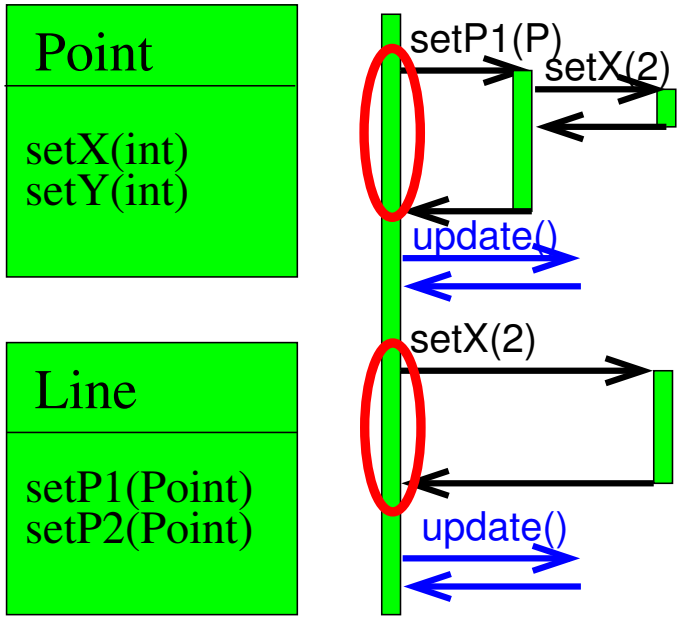



```
aspect updateDisplay{

  pointcut stateChanged() :
    || call(void Point.set*(..))
    || call(void Line.set*(..));

  after() : stateChanged() {

    display.update();
  }
}
```

AOP: Key Concepts

- Join points: where aspects intervene
- Pointcut: select join points
- Advice: what aspect does

Example in Java and AspectJ

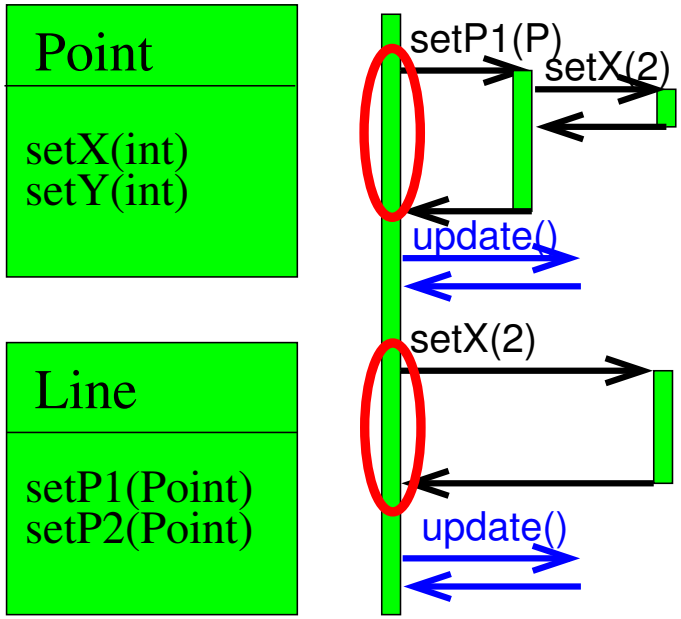



```
aspect updateDisplay{

  pointcut stateChanged() :
    || call(void Point.set*(..))
    || call(void Line.set*(..));

  after() : stateChanged()
    && !cflowbelow(stateChanged()){
    display.update();
  }
}
```

AOP: Key Concepts

- Join points: where aspects intervene
- Pointcut: select join points
- Advice: what aspect does

Example in Java and AspectJ




```
aspect updateDisplay{

  pointcut stateChanged() :
    || call(void Point.set*(..))
    || call(void Line.set*(..));

  after() : stateChanged()
    && !cflowbelow(stateChanged()){
    display.update();
  }
}
```

AOP: Key Concepts

- Join points: where aspects intervene
- Pointcut: select join points
- Advice: what aspect does

Conclusion on AspectJ

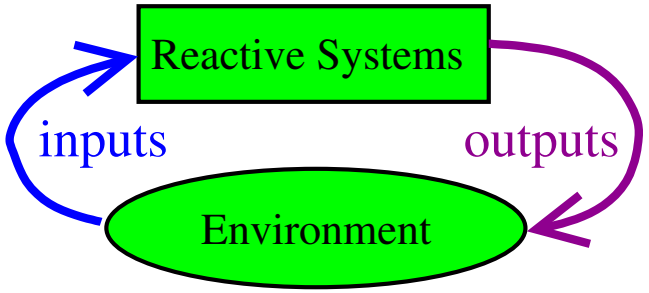
- Based on lexical elements
- Powerful constructs

Reactive Systems and Synchronous Languages

- Constant interaction with environment

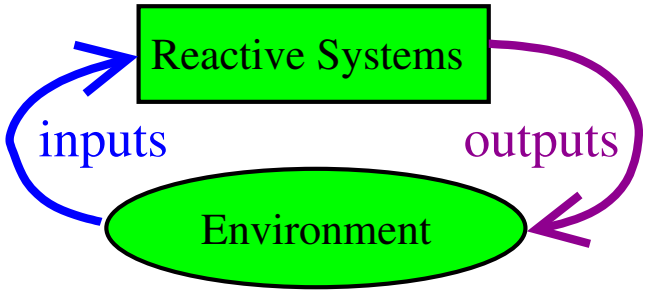
Reactive Systems and Synchronous Languages

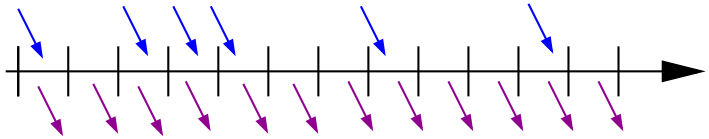
- Constant interaction with environment
- Receive inputs, emit outputs
- Often safety critical, need for formal semantics and verification



Reactive Systems and Synchronous Languages

- Constant interaction with environment
- Receive inputs, emit outputs
- Often safety critical, need for formal semantics and verification
- Synchronous languages: simple semantics, discrete time



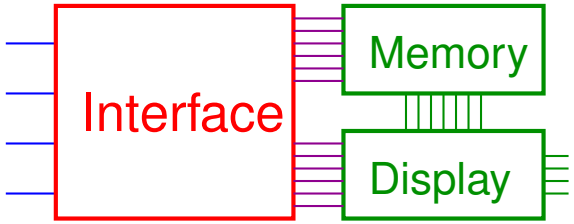


A Wristwatch – A Reactive System

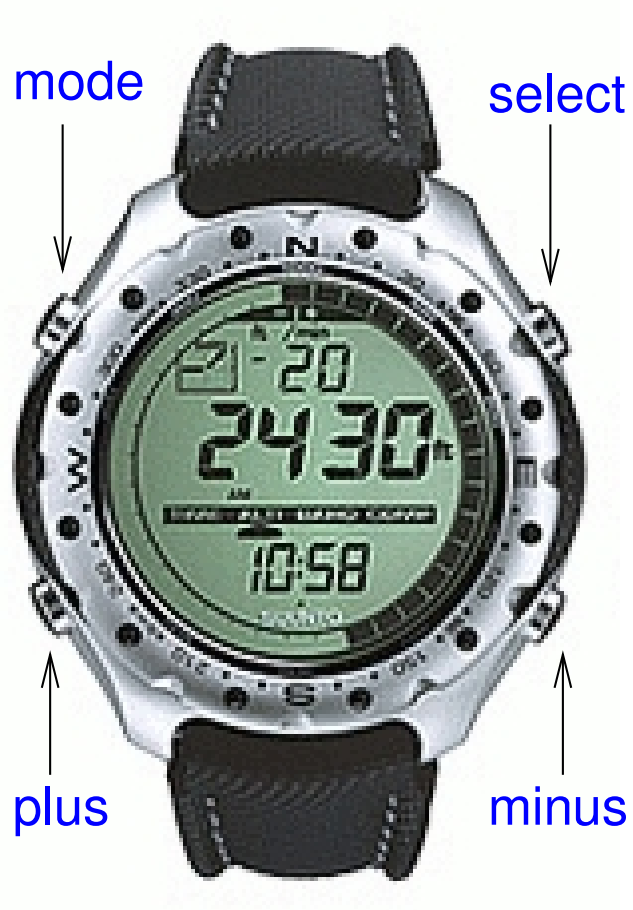
- Wristwatch with four buttons
- Two Models
  - Altimax: watch, altimeter, barometer
  - Vector: Altimax + compass

A Wristwatch – A Reactive System

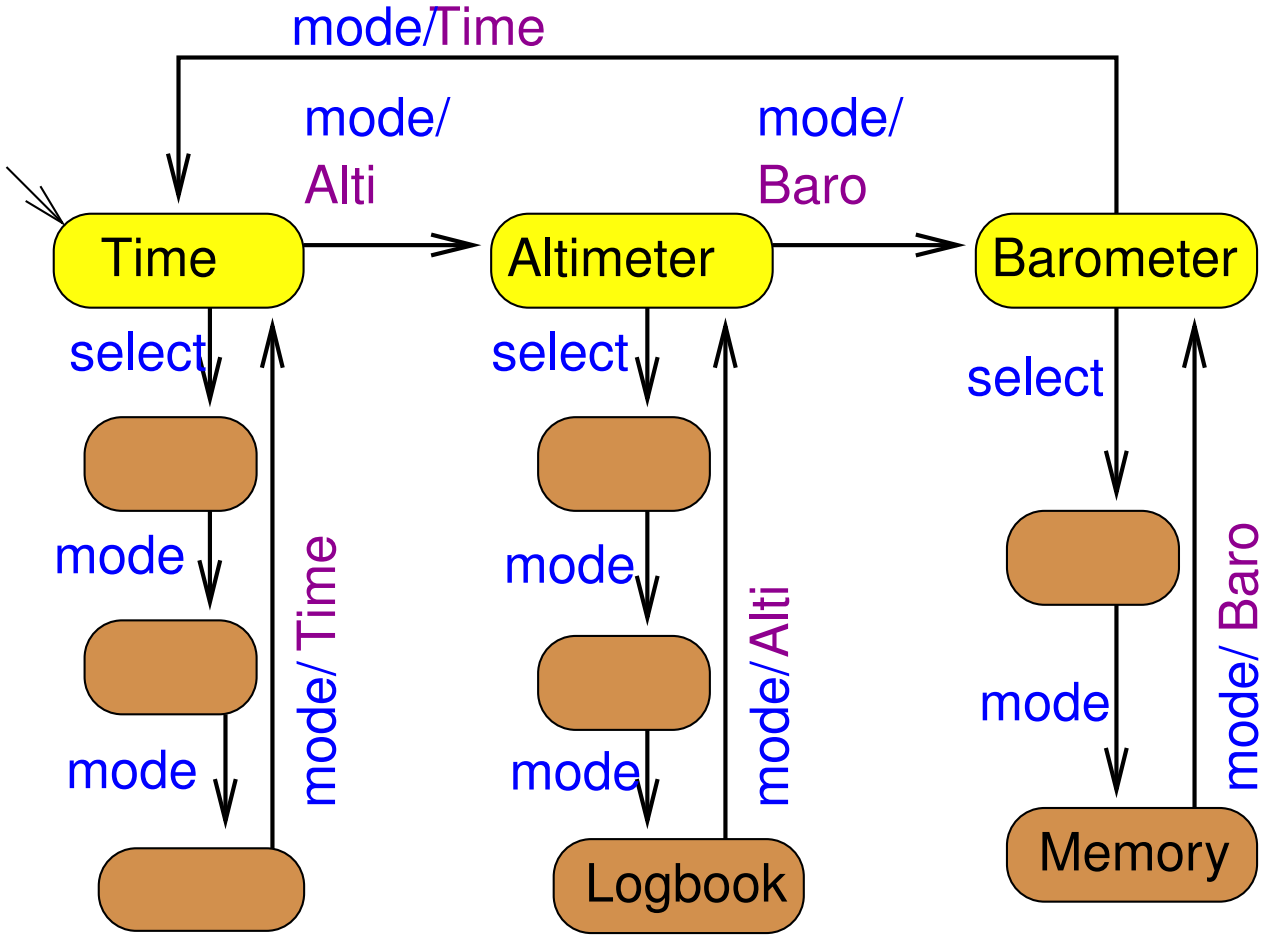
- Wristwatch with four buttons
- Two Models
  - Altimax: watch, altimeter, barometer
  - Vector: Altimax + compass
- We model interface component



- Inputs: buttons
- Outputs: signals to other components

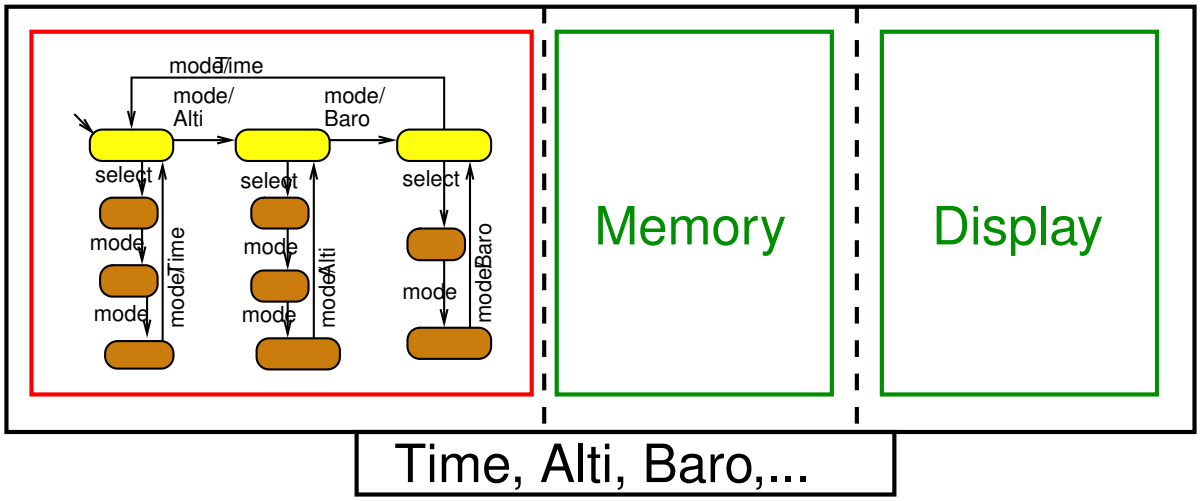


The Altimax Interface



Argos, a Synchronous Language

- Base element: Mealy automata
- Arrange modules in parallel
- Modules communicate with local signals

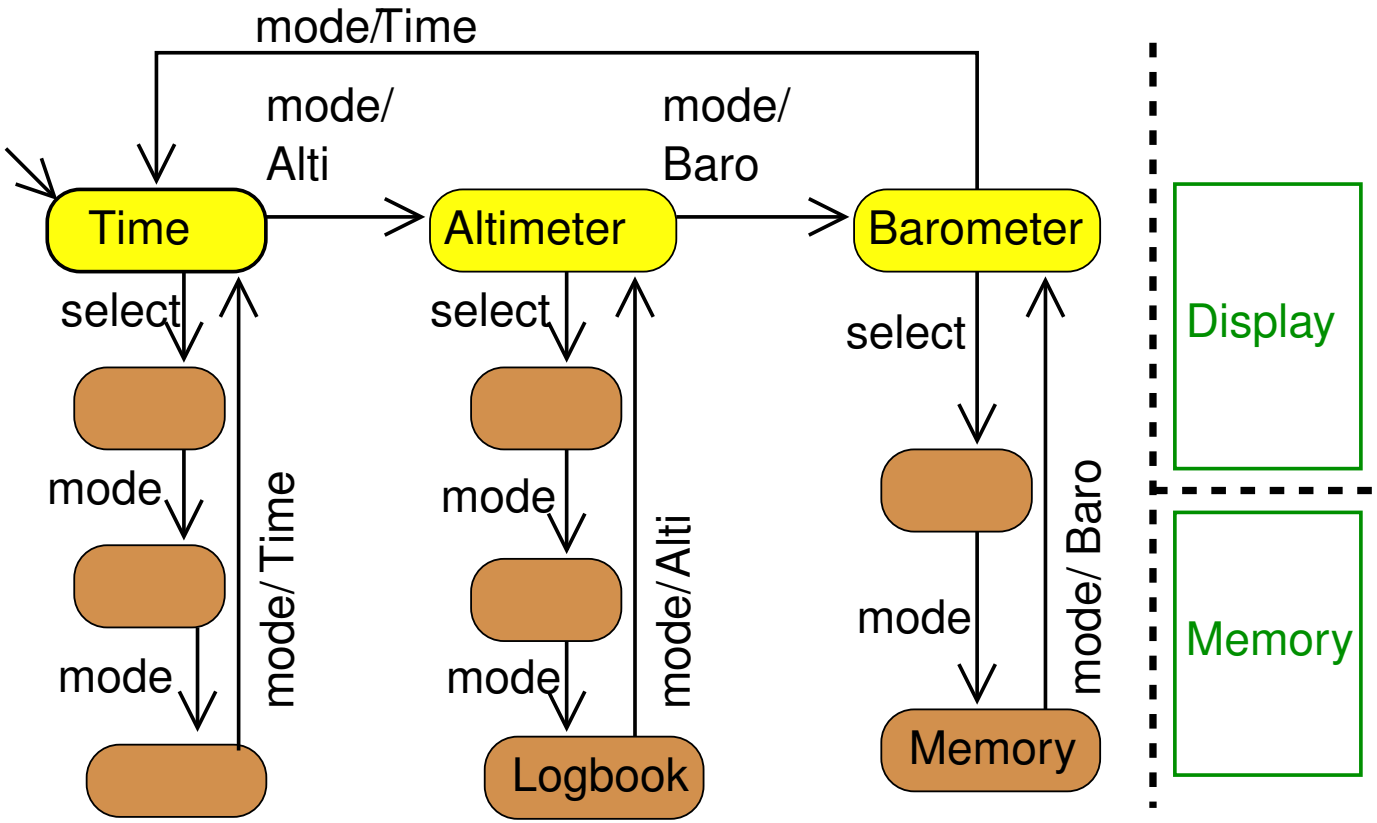


Crosscutting Concern 1: Shortcut

- Crosscutting concerns also in reactive systems?

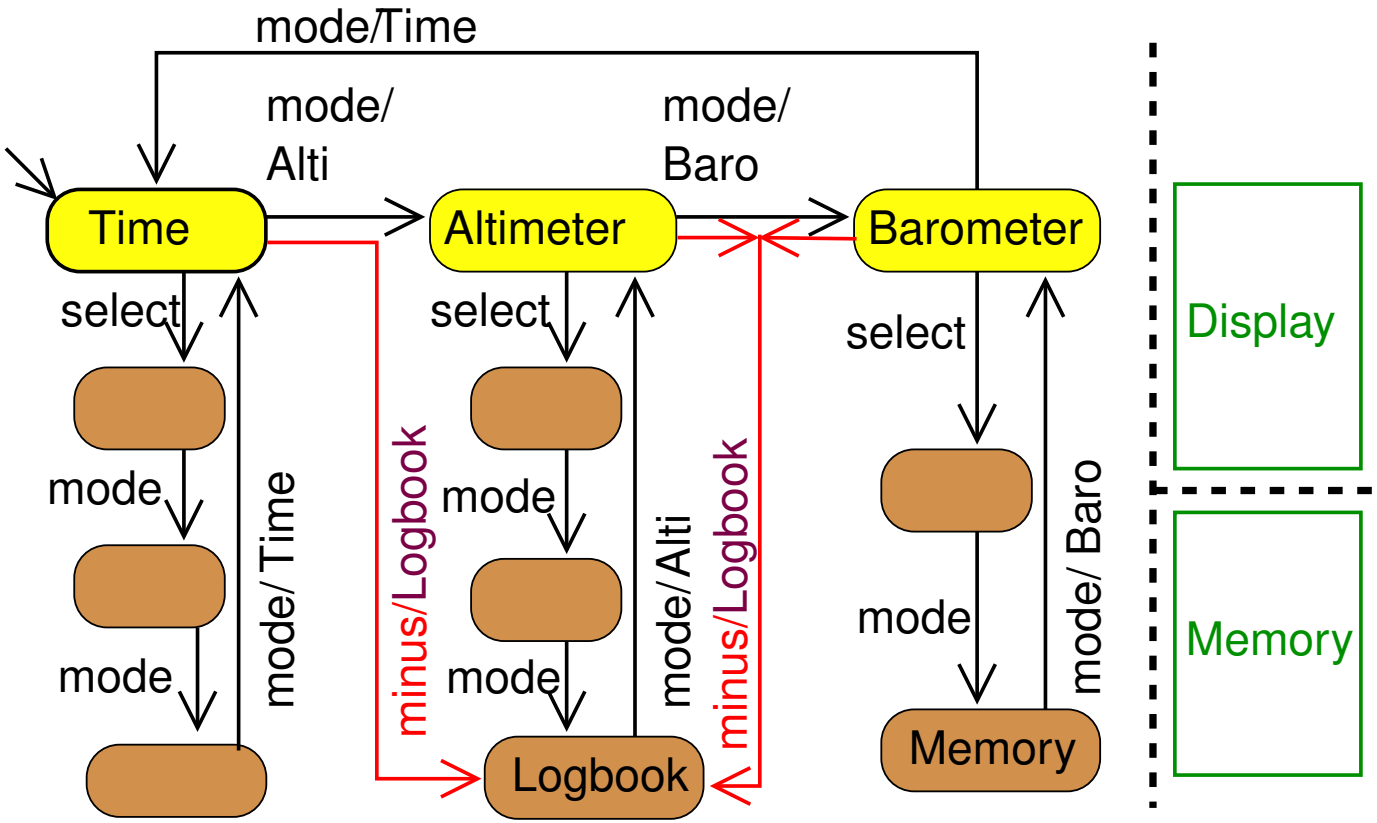
Crosscutting Concern 1: Shortcut

- Crosscutting concerns also in reactive systems?
- Example from watch: minus button not used in main modes
- Use it to jump directly to Logbook mode



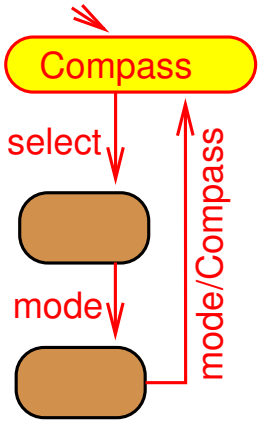
Crosscutting Concern 1: Shortcut

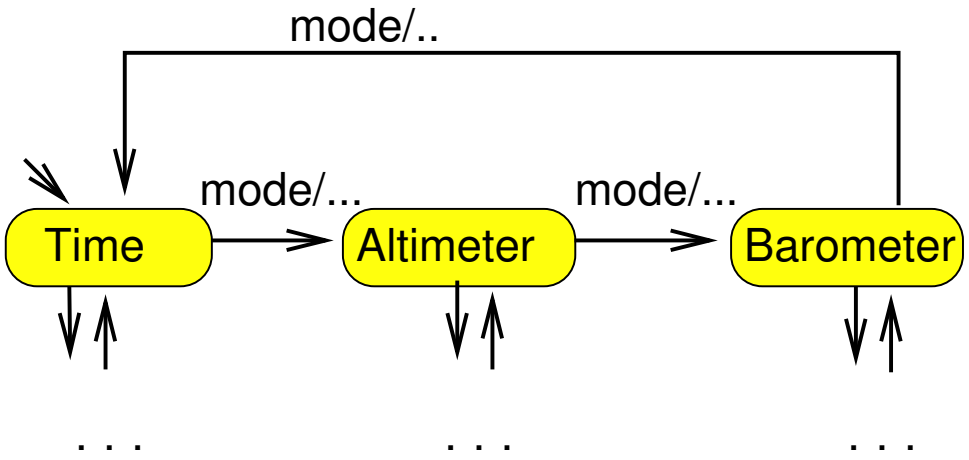
- Crosscutting concerns also in reactive systems?
- Example from watch: minus button not used in main modes
- Use it to jump directly to Logbook mode

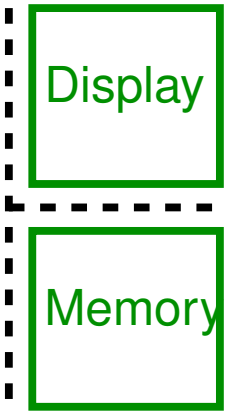


Crosscutting Concern 2: Compass Mode

- Vector model has compass mode
- Add Compass mode to Altimax base program

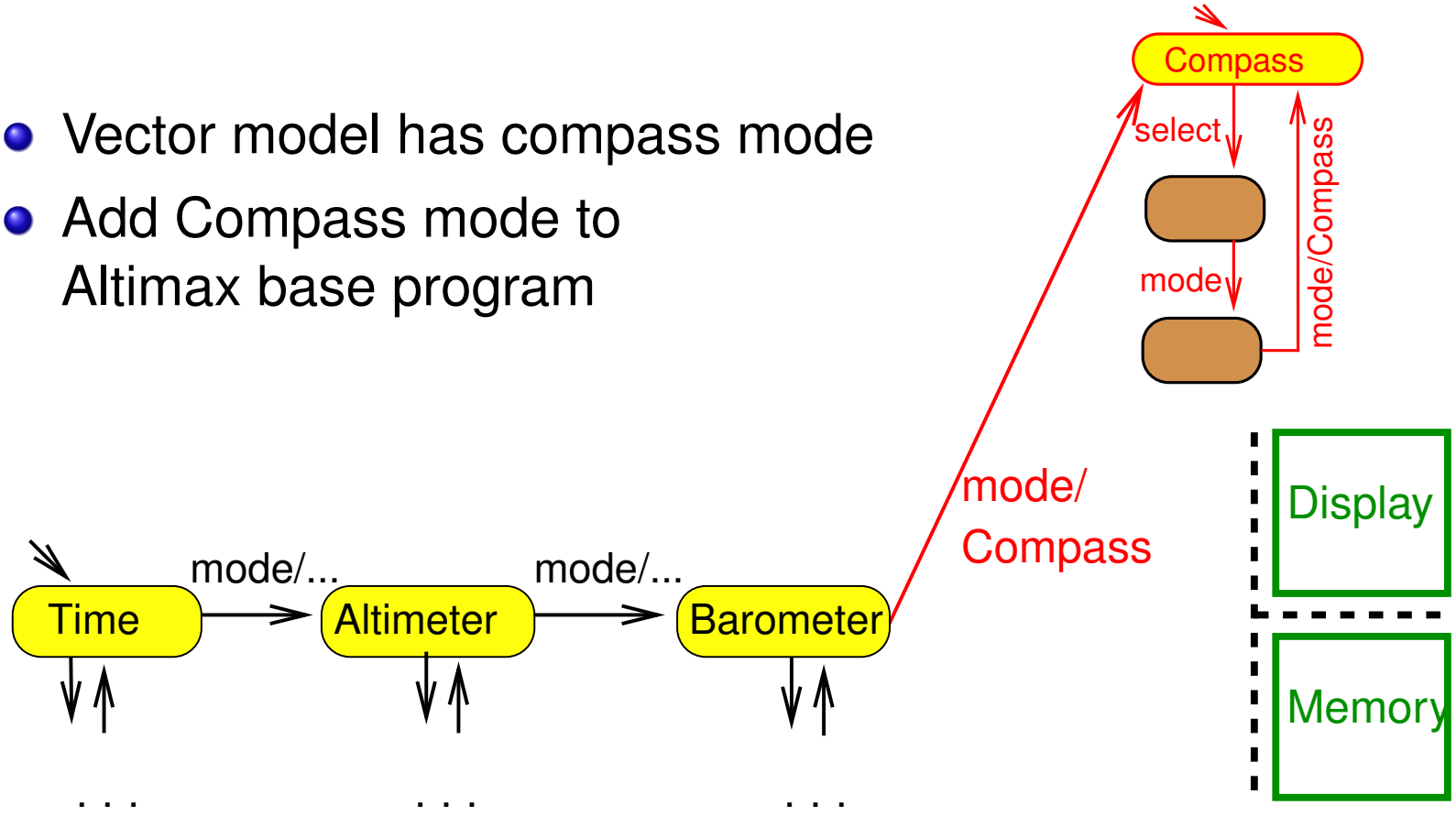






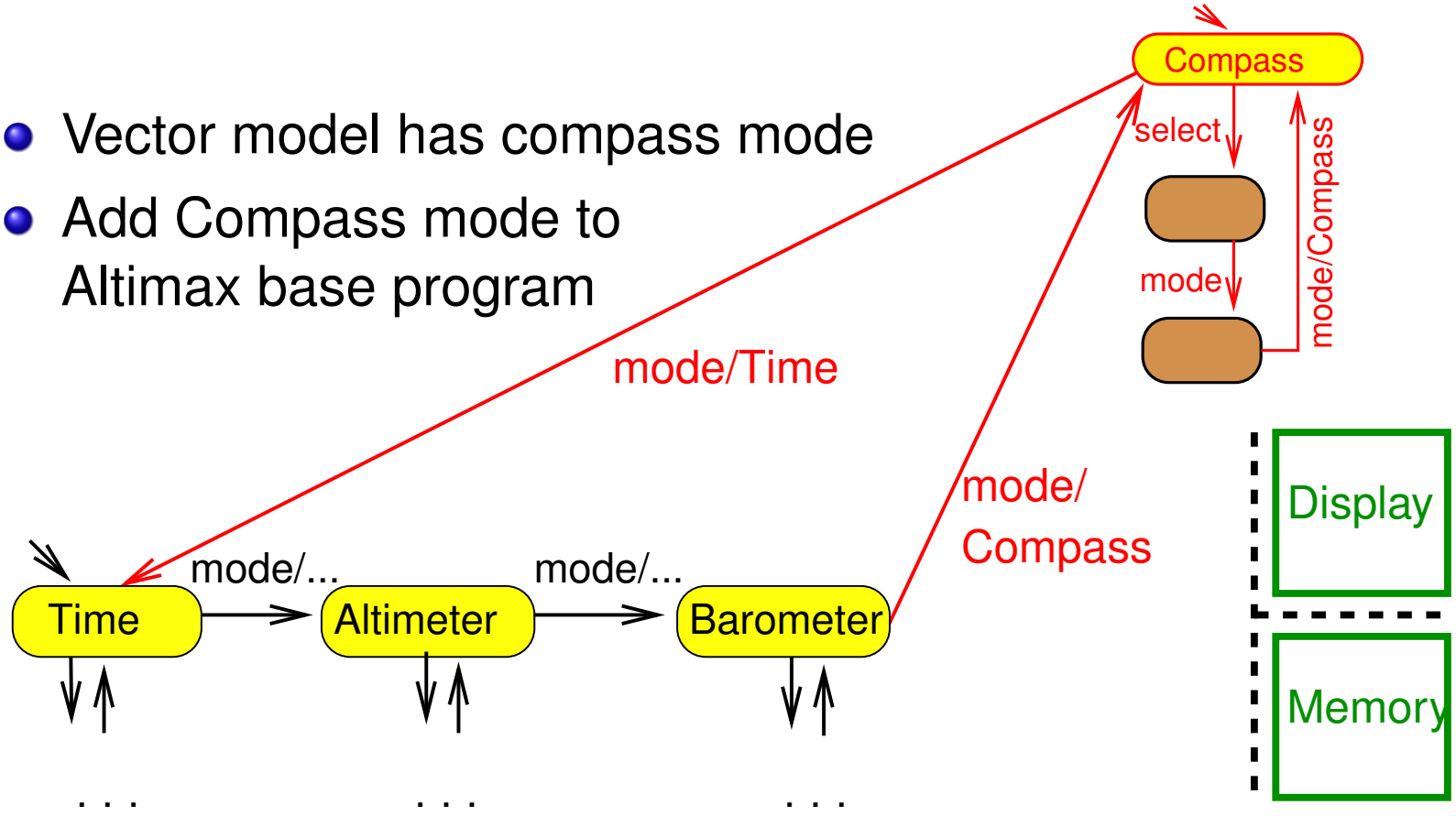
Crosscutting Concern 2: Compass Mode

- Vector model has compass mode
- Add Compass mode to Altimax base program



Crosscutting Concern 2: Compass Mode

- Vector model has compass mode
- Add Compass mode to Altimax base program



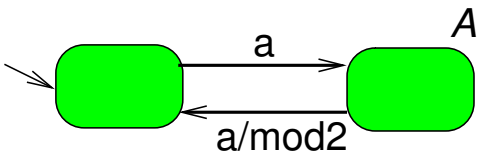
Larissa: Aspects for Argos

- Goal: aspect language for synchronous languages
- Argos adequate base language
  - simple synchronous language
  - expressive, characterizing constructs
- Must express cross-cutting concerns
- Same concepts as other aspect languages:
  join points, pointcuts, advice

Outline

Argos Operators

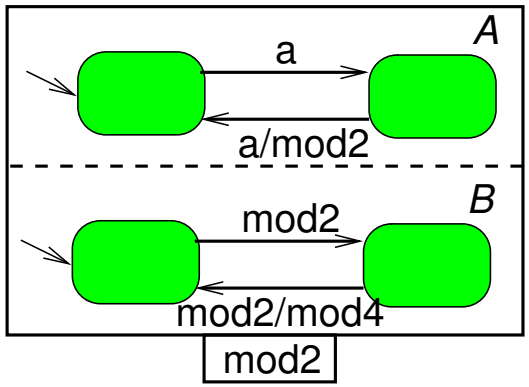
- Base elements: boolean signals,
  complete and deterministic Mealy automata



A

Argos Operators

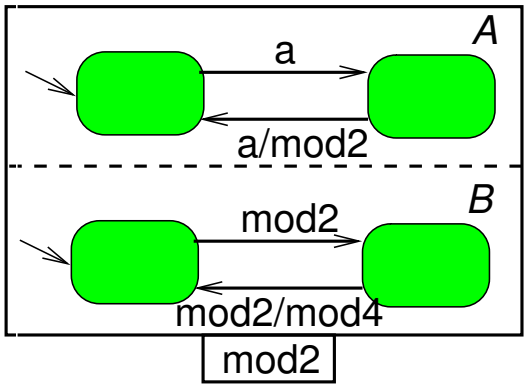
- Base elements: boolean signals, complete and deterministic Mealy automata
- Main operators: parallel product, local signals

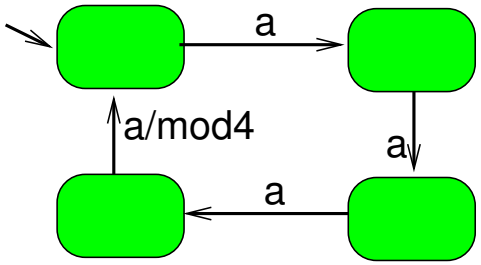


$$(A \| B) \setminus \{\text{mod2}\}$$

Argos Operators

- Base elements: boolean signals, complete and deterministic Mealy automata
- Main operators: parallel product, local signals
- Semantics: compilation into flat automata





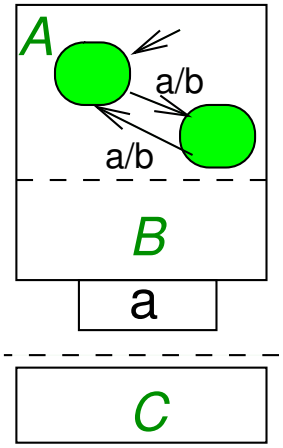
$(A \| B) \setminus \{\text{mod2}\}$

Encapsulation

- Argos programs form expressions, e.g.

$$((A \| B) \setminus \{a\}) \ \| \ C$$

- A, B and C automata or Argos expressions

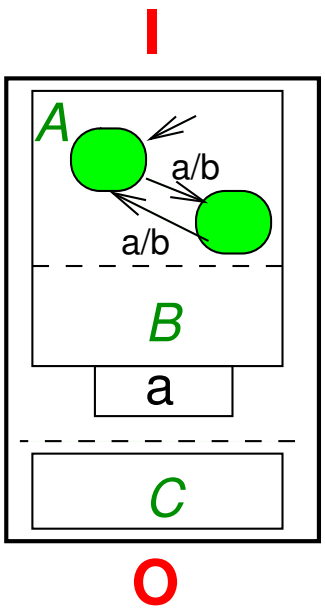


Encapsulation

- Argos programs form expressions, e.g.

$$((A \| B) \setminus \{a\}) \ \| \ C$$

- A, B and C automata or Argos expressions
- Interface: inputs, outputs

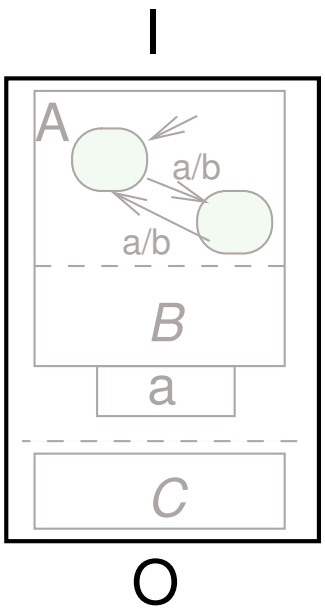


Encapsulation

- Argos programs form expressions, e.g.

$$((A \| B) \setminus \{a\}) \ \| \ C$$

- A, B and C automata or Argos expressions
- Interface: inputs, outputs
- Strong encapsulation: component structure invisible from outside



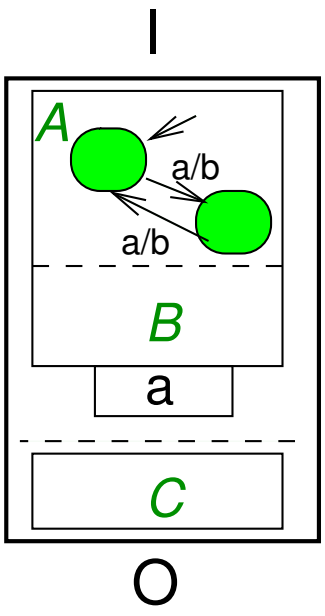
Encapsulation

- Argos programs form expressions, e.g.

$$((A \| B) \setminus \{a\}) \ \| \ C$$

- A, B and C automata or Argos expressions
- Interface: inputs, outputs
- Strong encapsulation: component structure invisible from outside
- Operators preserve i/o-trace equivalence ($\sim$): if $A' \sim A$, then

$$((A' \| B) \setminus \{a\}) \| C \ \sim \ ((A \| B) \setminus \{a\}) \| C$$



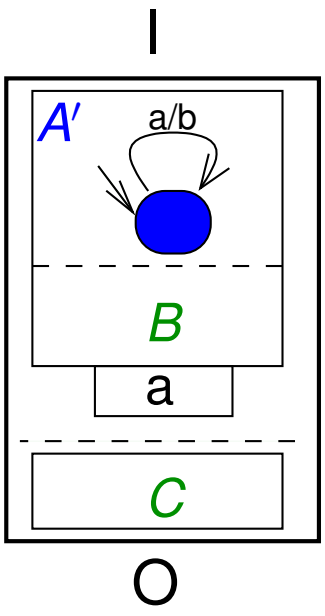
Encapsulation

- Argos programs form expressions, e.g.

$$((A \| B) \setminus \{a\}) \ \| \ C$$

- A, B and C automata or Argos expressions
- Interface: inputs, outputs
- Strong encapsulation: component structure invisible from outside
- Operators preserve i/o-trace equivalence ($\sim$): if $A' \sim A$, then

$$((A' \| B) \setminus \{a\}) \| C \ \sim \ ((A \| B) \setminus \{a\}) \| C$$



Requirements: Aspects for Argos

Express cross-cutting concerns

- crosscut structure of Argos expressions
- parallel composition can express some aspects for sequential languages

Requirements: Aspects for Argos

Express cross-cutting concerns

- crosscut structure of Argos expressions
- parallel composition can express some aspects for sequential languages

Integrate well into Argos

- define as translation into automaton
- simple, formal semantics
- respect encapsulation, as other Argos operators
  - usually not respected by aspect languages

Related Work

Aspects and Parallelism

- Concurrent aspects [Douence et al, GPCE06]:
  - asynchronous base program, asynchronous execution of advice

Related Work

Aspects and Parallelism

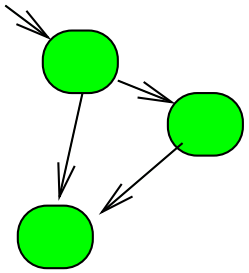
- Concurrent aspects [Douence et al, GPCE06]:
  - asynchronous base program, asynchronous execution of advice

Formal Properties

- Many formalisations of aspect languages
- Aspects preserving the encapsulation:
  - Composition Filters [Bergmans, Aksit]: intercept and modify messages between components
  - Open Modules [Aldrich, ECOOP06]: add additional information to interface

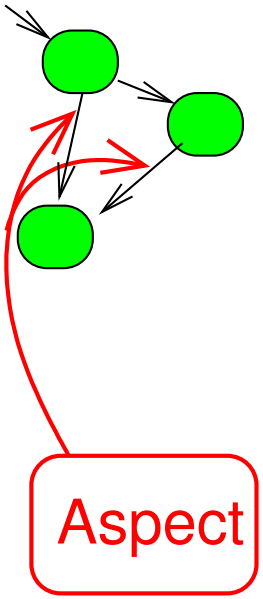
Larissa

- Join points: one step in the execution
- I.e., transitions in an automaton



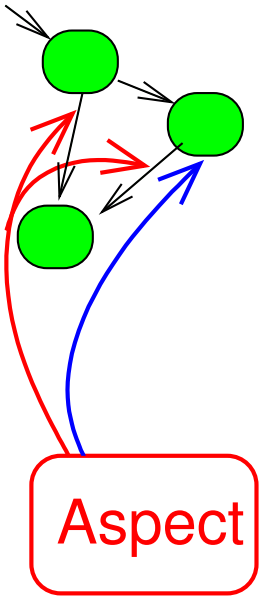
Larissa

- Join points: one step in the execution
- I.e., transitions in an automaton
- Pointcut: select transitions in automaton



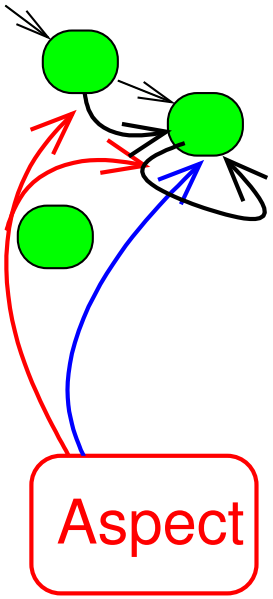
Larissa

- Join points: one step in the execution
- I.e., transitions in an automaton
- Pointcut: select transitions in automaton
- Advice: modify transitions
  - change target state and outputs



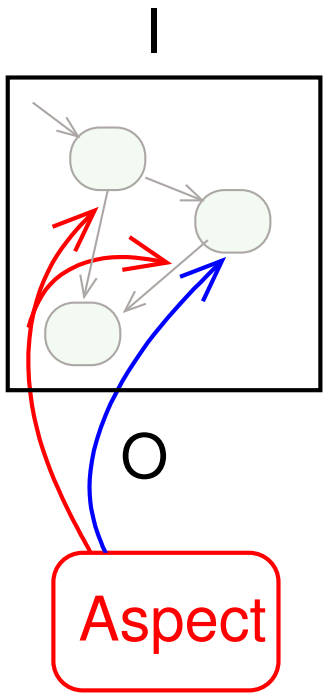
Larissa

- Join points: one step in the execution
- I.e., transitions in an automaton
- Pointcut: select transitions in automaton
- Advice: modify transitions
  - change target state and outputs



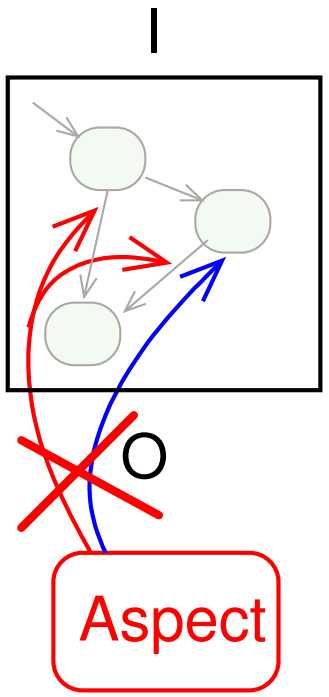
Larissa

- Join points: one step in the execution
- I.e., transitions in an automaton
- Pointcut: select transitions in automaton
- Advice: modify transitions
  - change target state and outputs
- Challenge: respect encapsulation



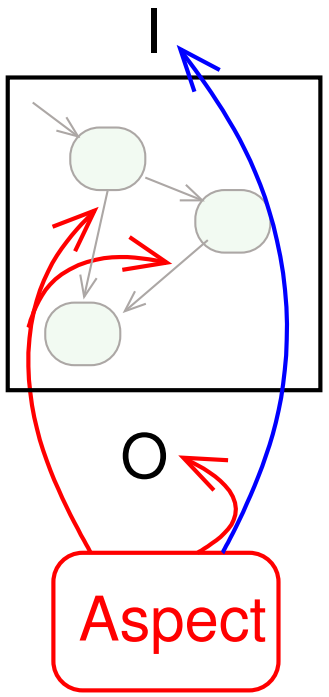
Larissa

- Join points: one step in the execution
- I.e., transitions in an automaton
- Pointcut: select transitions in automaton
- Advice: modify transitions
  - change target state and outputs
- Challenge: respect encapsulation



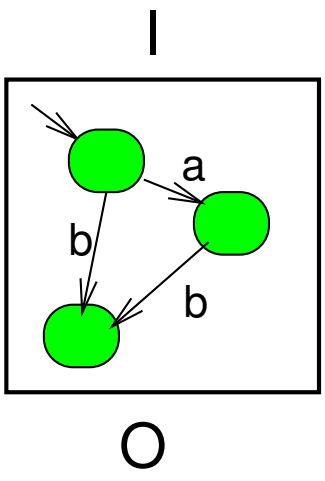
Larissa

- Join points: one step in the execution
- I.e., transitions in an automaton
- Pointcut: select transitions in automaton
- Advice: modify transitions
  - change target state and outputs
- Challenge: respect encapsulation
  - aspect must only refer to interface



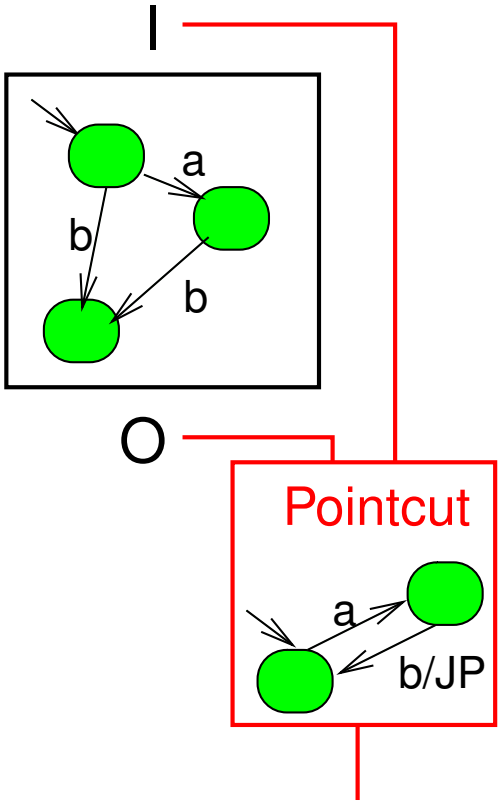
Pointcuts

- Must select transitions



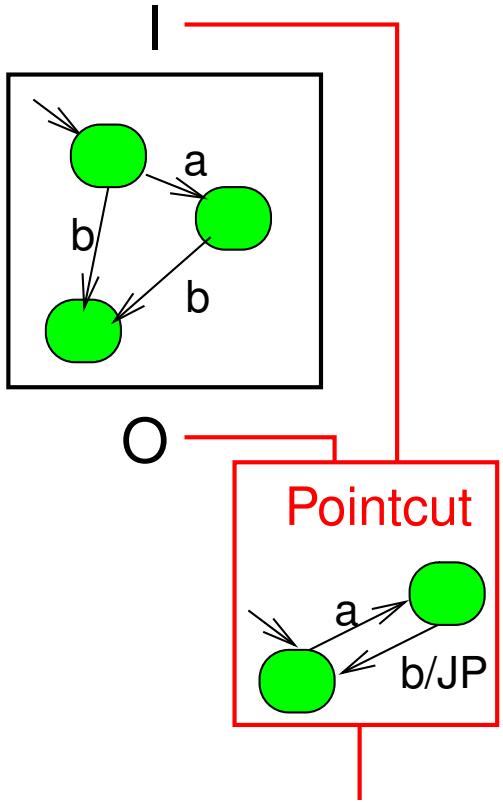
Pointcuts

- Must select transitions
- Solution: observer automaton
  - inputs: inputs and outputs of observed program
  - one output JP



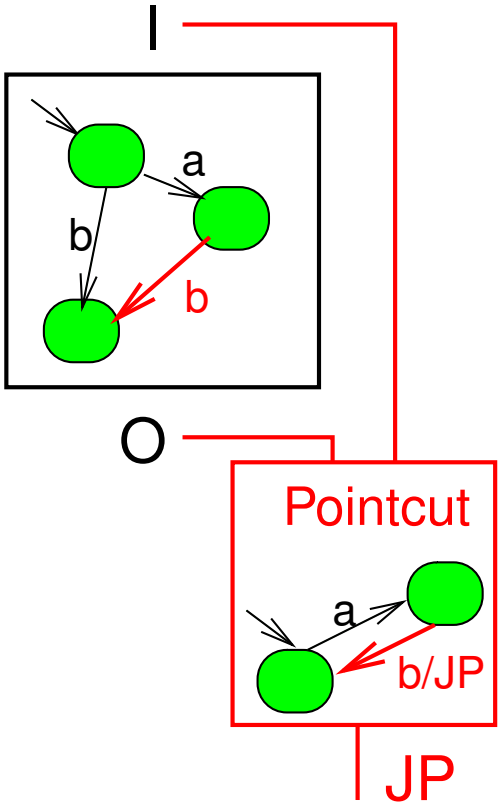
Pointcuts

- Must select transitions
- Solution: observer automaton
  - inputs: inputs and outputs of observed program
  - one output JP
- Pointcut emits JP
  $\Rightarrow$ transition in program selected



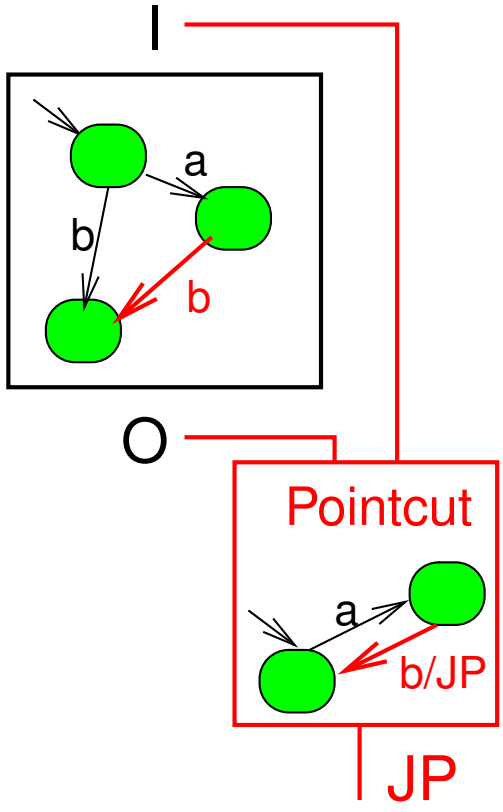
Pointcuts

- Must select transitions
- Solution: observer automaton
  - inputs: inputs and outputs of observed program
  - one output JP
- Pointcut emits JP
  $\Rightarrow$ transition in program selected



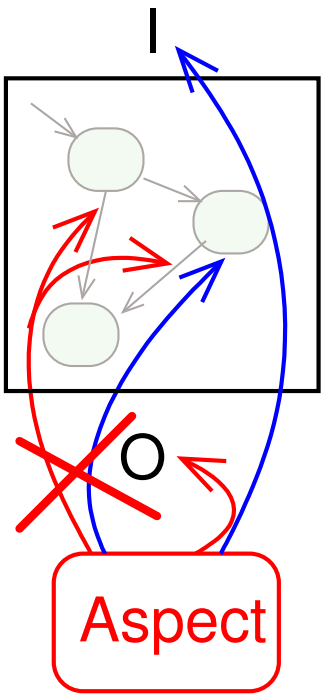
Pointcuts

- Must select transitions
- Solution: observer automaton
  - inputs: inputs and outputs of observed program
  - one output JP
- Pointcut emits JP
  $\Rightarrow$ transition in program selected
- Transitions identified statically by parallel product



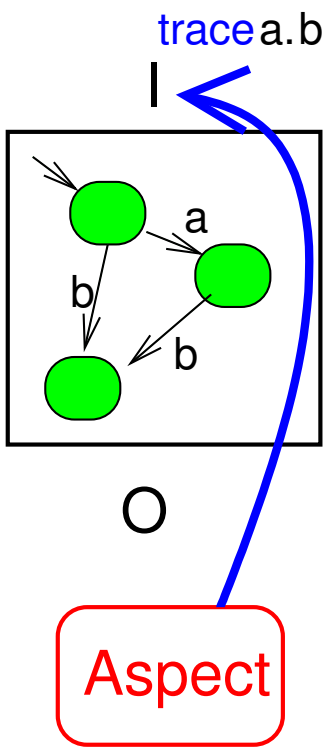
Advice

- Difficulty: specify one new target state



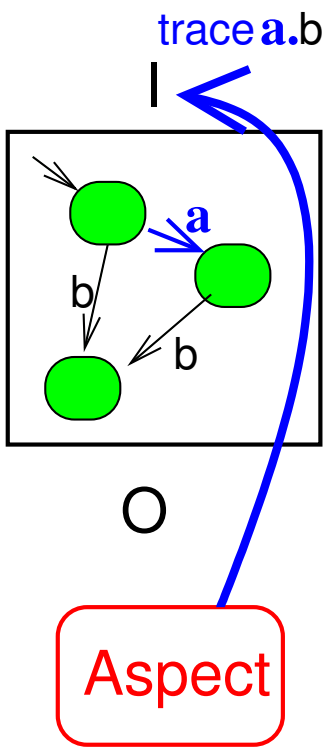
Advice

- Difficulty: specify one new target state
- Solution: execute finite input trace
  - automaton deterministic, thus always identifies one state



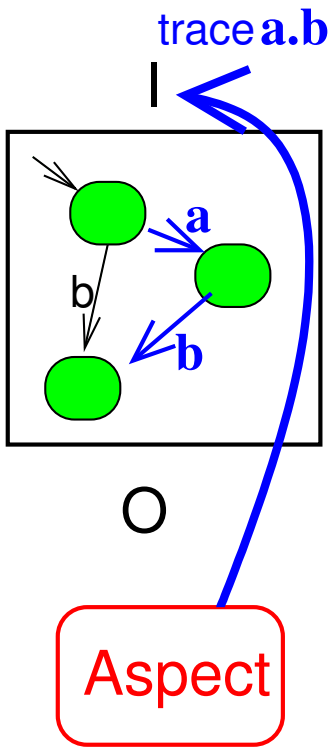
Advice

- Difficulty: specify one new target state
- Solution: execute finite input trace
  - automaton deterministic, thus always identifies one state



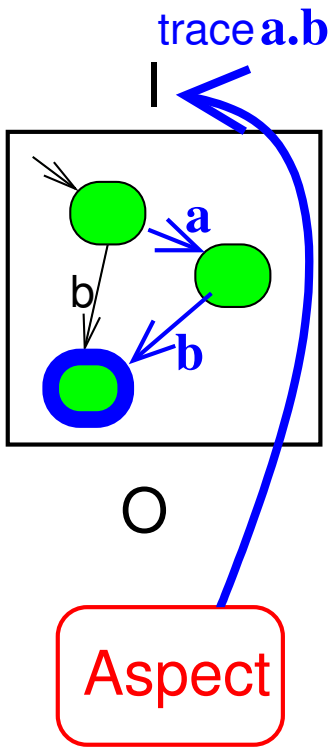
Advice

- Difficulty: specify one new target state
- Solution: execute finite input trace
  - automaton deterministic, thus always identifies one state



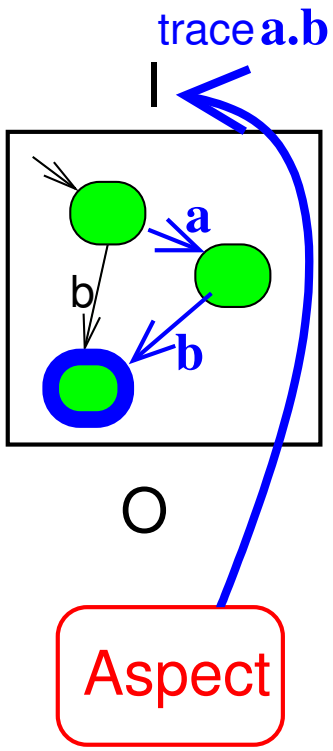
Advice

- Difficulty: specify one new target state
- Solution: execute finite input trace
  - automaton deterministic, thus always identifies one state



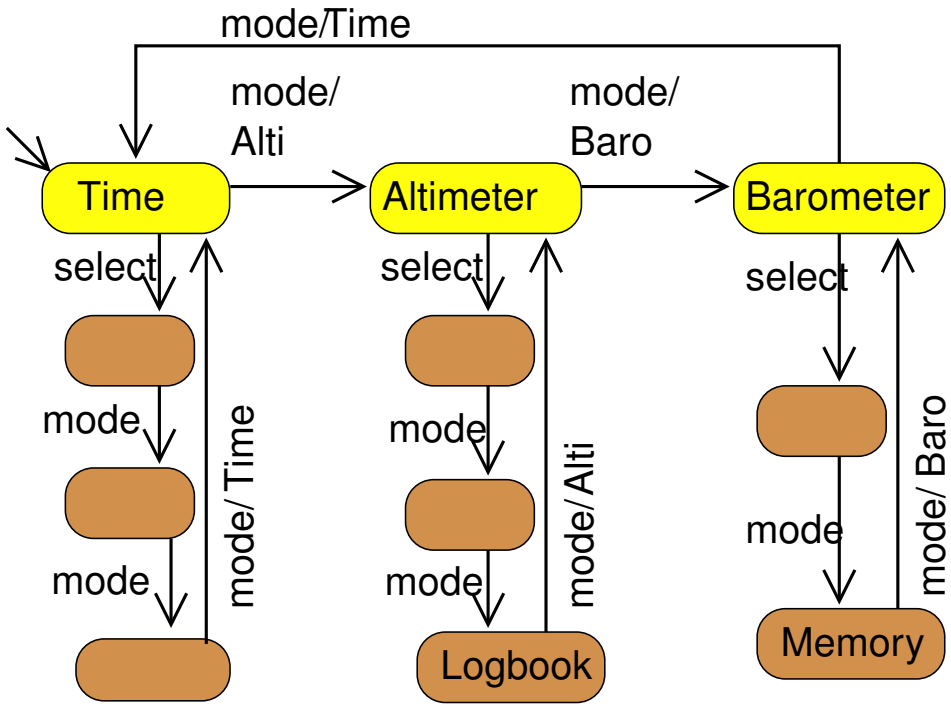
Advice

- Difficulty: specify one new target state
- Solution: execute finite input trace
  - automaton deterministic, thus always identifies one state
- Two kinds:
  - toInit advice: execute trace from initial state
  - toCurrent advice: execute trace from source state of transition



Example: Logbook Shortcut Aspect LB

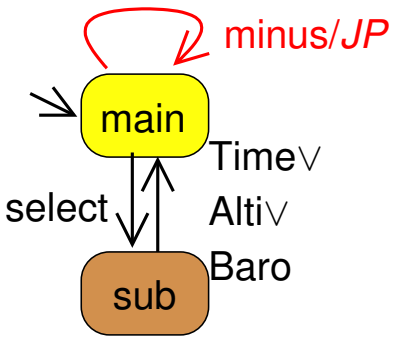
- Pointcut: transitions in main modes where minus is true
- Advice: trace mode.select.mode.mode, output Logbook



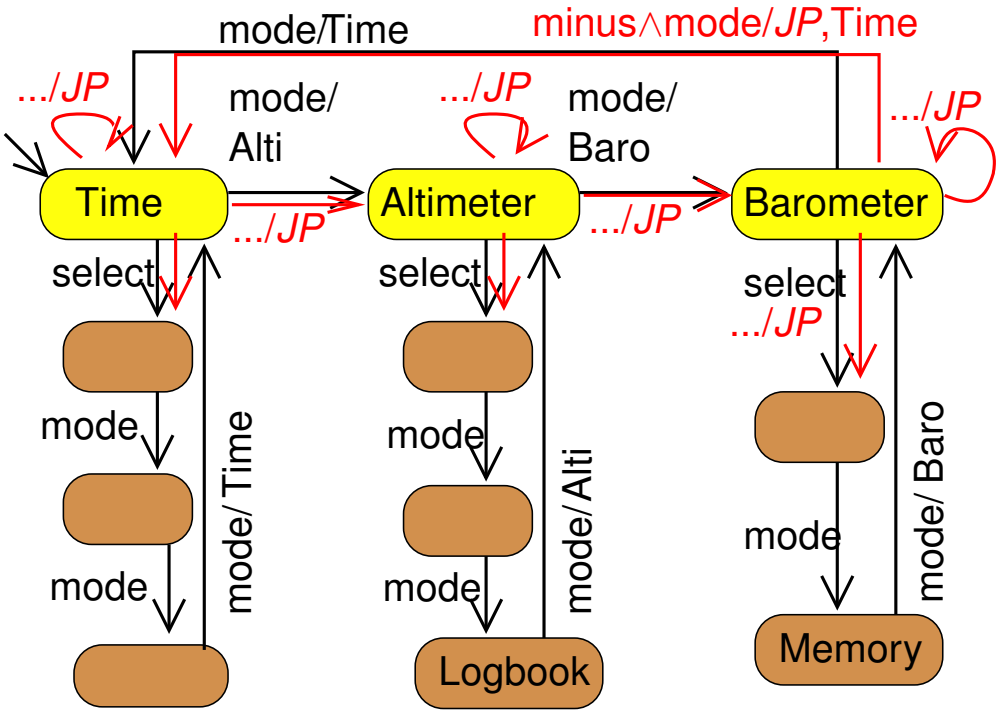
altimax

Example: Logbook Shortcut Aspect LB

- Pointcut: transitions in main modes where minus is true
- Advice: trace mode.select.mode.mode, output Logbook



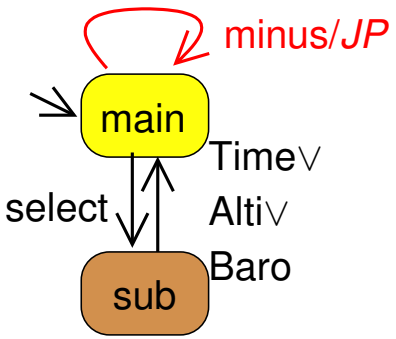
pointcut of LB



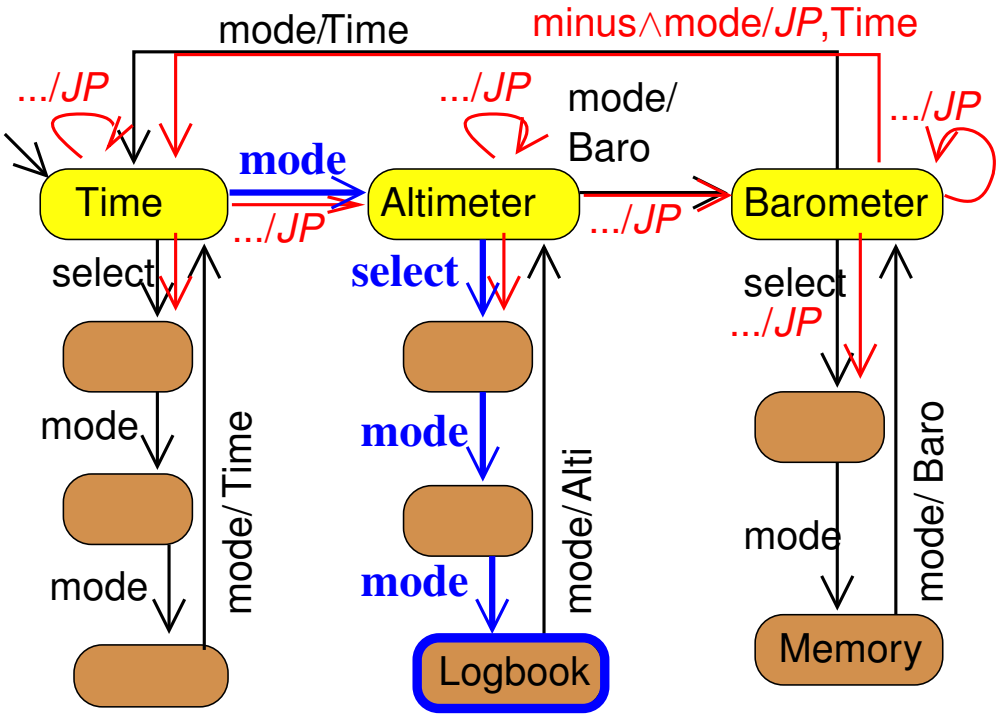
altimax

Example: Logbook Shortcut Aspect LB

- Pointcut: transitions in main modes where minus is true
- Advice: trace mode.select.mode.mode, output Logbook



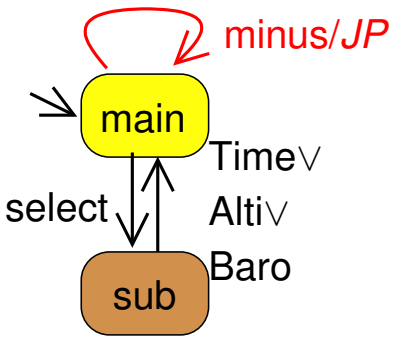
pointcut of LB



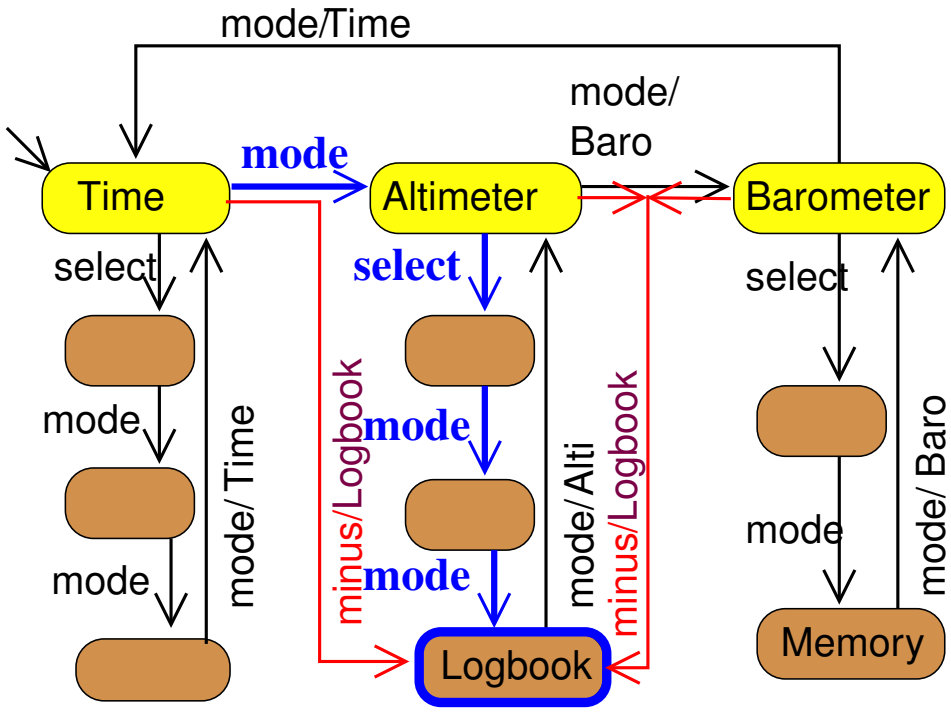
altimax

Example: Logbook Shortcut Aspect LB

- Pointcut: transitions in main modes where minus is true
- Advice: trace mode.select.mode.mode, output Logbook



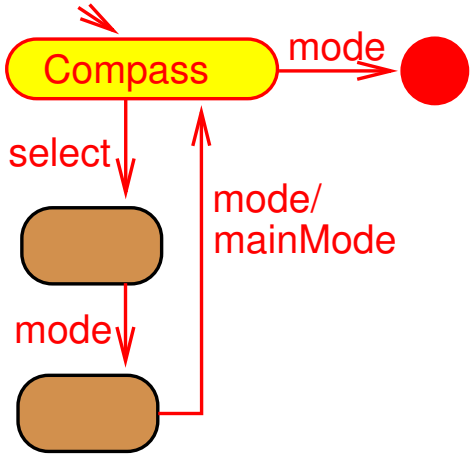
pointcut of LB



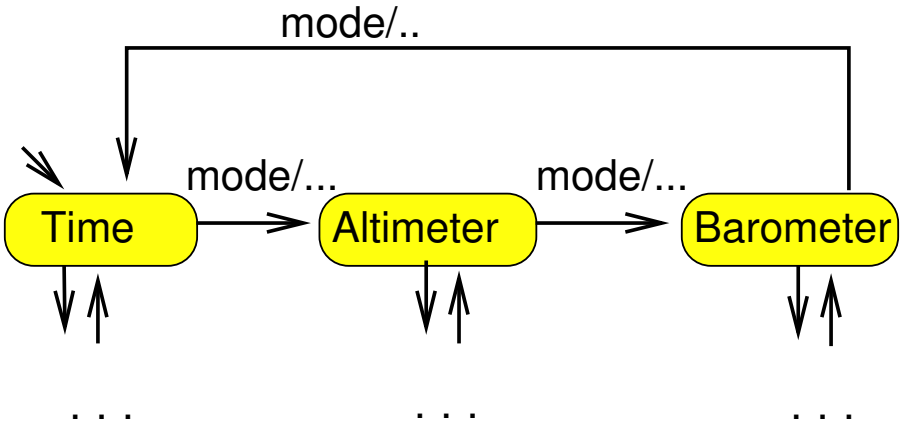
altimax◁LB

Advice Program

- Advice insufficient for Compass Concern
- Replace transition by advice program
- Advice program has terminating state: represents return to base program

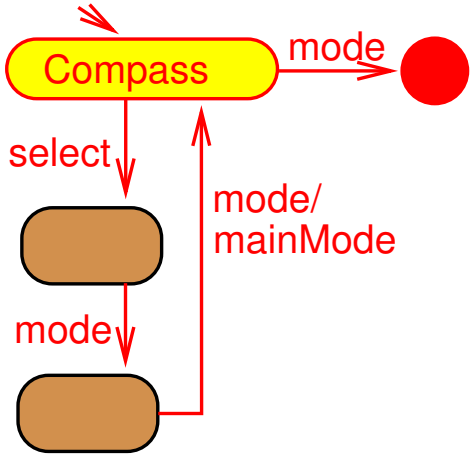


advice program

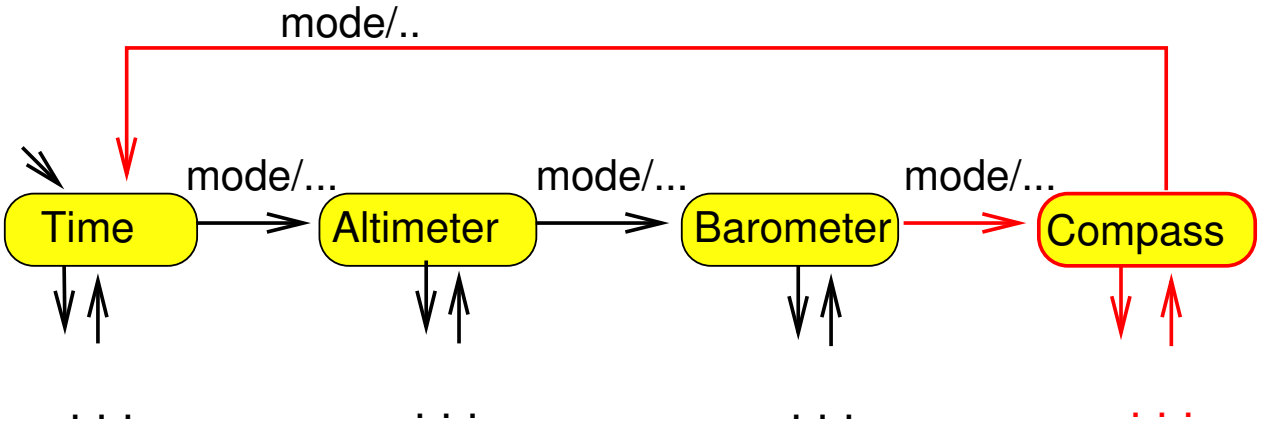


Advice Program

- Advice insufficient for Compass Concern
- Replace transition by advice program
- Advice program has terminating state: represents return to base program



advice program



altimax ◁compass

Recovery Advice and Compiler

Recovery advice

- “Jumping backward”
- Identify set of recovery states
- Jump to last recovery state that was passed

Recovery Advice and Compiler

Recovery advice

- "Jumping backward"
- Identify set of recovery states
- Jump to last recovery state that was passed

Compiler for Argos and Larissa

- All language variants implemented
- Experimentation with many examples
- Written in Java, AspectJ, BDD library
- Available at

`http://www-verimag.imag.fr/~stauch/ArgosCompiler/`

Outline

Formal Analysis Tools

- Larissa: small language, formally defined, with simple semantics
- Well adapted to study formal properties of aspect languages
- We studied two such properties:
  - interaction of several aspects
  - combination of Larissa with contracts

Aspect Interaction

Aspect Interaction

- Do several aspects influence each other?
- When is $P \triangleleft A1 \triangleleft A2 \sim P \triangleleft A2 \triangleleft A1$?

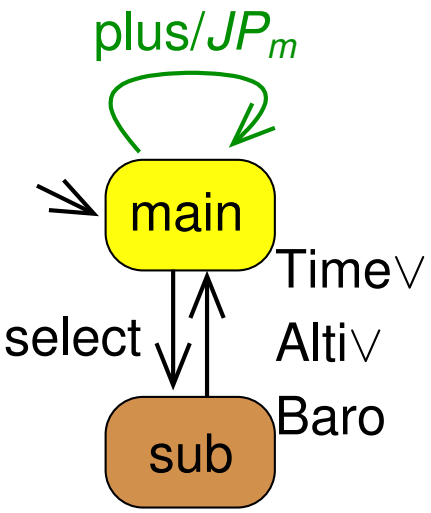
Aspect Interaction

Aspect Interaction

- Do several aspects influence each other?
- When is $P \triangleleft A1 \triangleleft A2 \sim P \triangleleft A2 \triangleleft A1$?

Example: Second Shortcut Aspect M

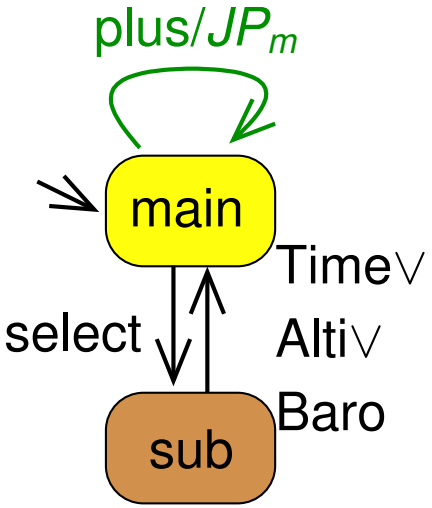
- Also use plus button as shortcut in the main modes
- Pressing plus goes to the Memory mode



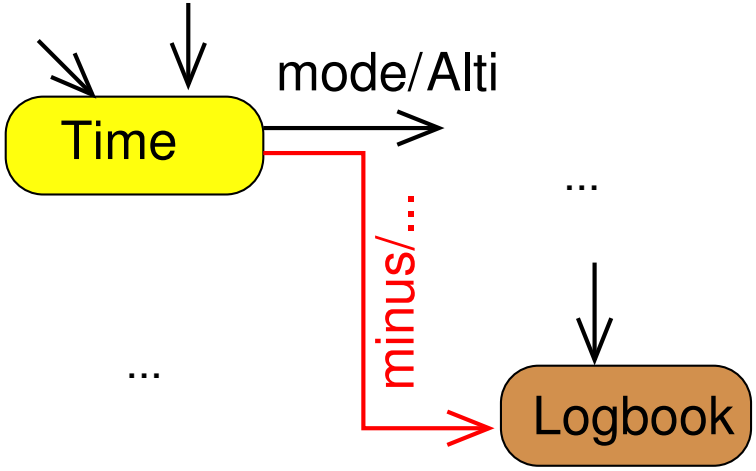
Pointcut of M

Weaving the Second Shortcut Aspect

- Weave M into altimax◁LB



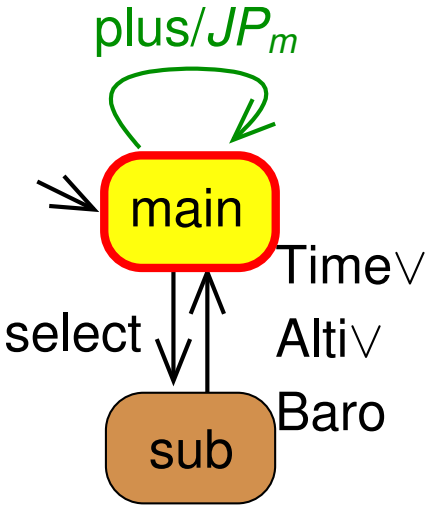
Pointcut of M



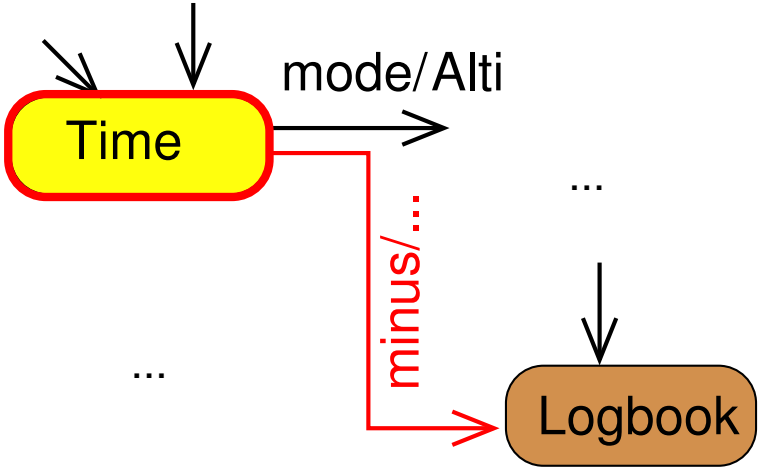
altimax◁LB

Weaving the Second Shortcut Aspect

- Weave M into altimax◁LB
- When pressing minus in main mode:
  - altimax◁LB goes to submode
  - pointcut stays in main mode



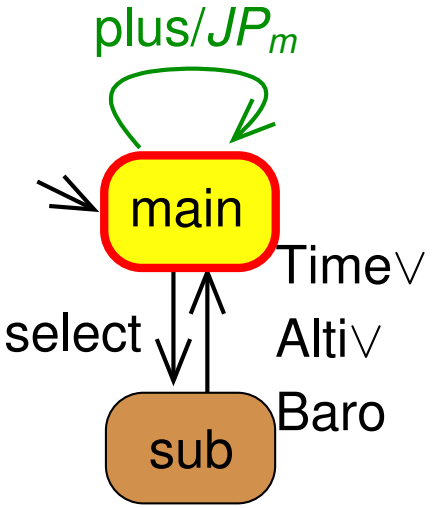
Pointcut of M



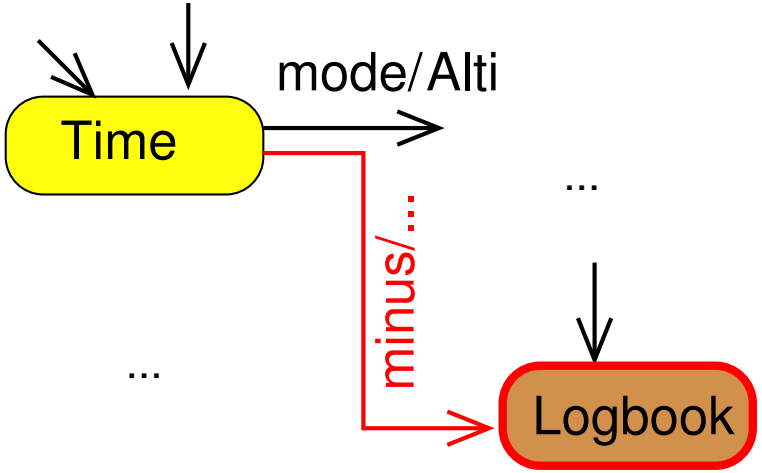
altimax◁LB

Weaving the Second Shortcut Aspect

- Weave M into altimax◁LB
- When pressing minus in main mode:
  - altimax◁LB goes to submode
  - pointcut stays in main mode



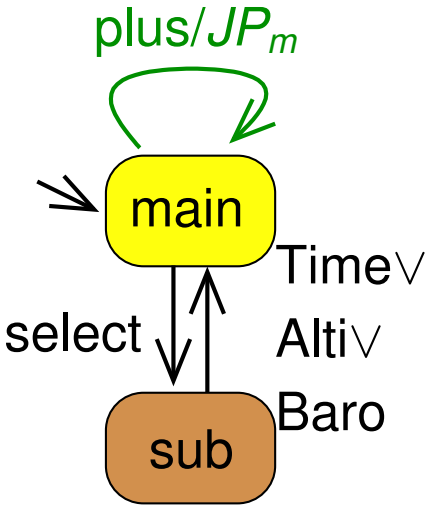
Pointcut of M



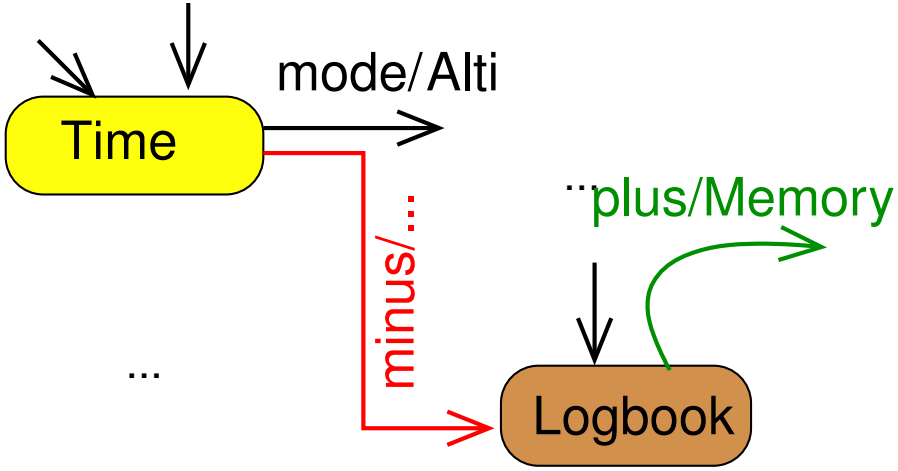
altimax◁LB

Weaving the Second Shortcut Aspect

- Weave M into altimax◁LB
- When pressing minus in main mode:
  - altimax◁LB goes to submode
  - pointcut stays in main mode
- Error: Advice transitions added to Logbook mode



Pointcut of M



altimax◁LB◁M

Joint Weaving

- Problem: aspect M written for altimax, not for altimax◁LB
- Idea: weave aspects jointly into the program

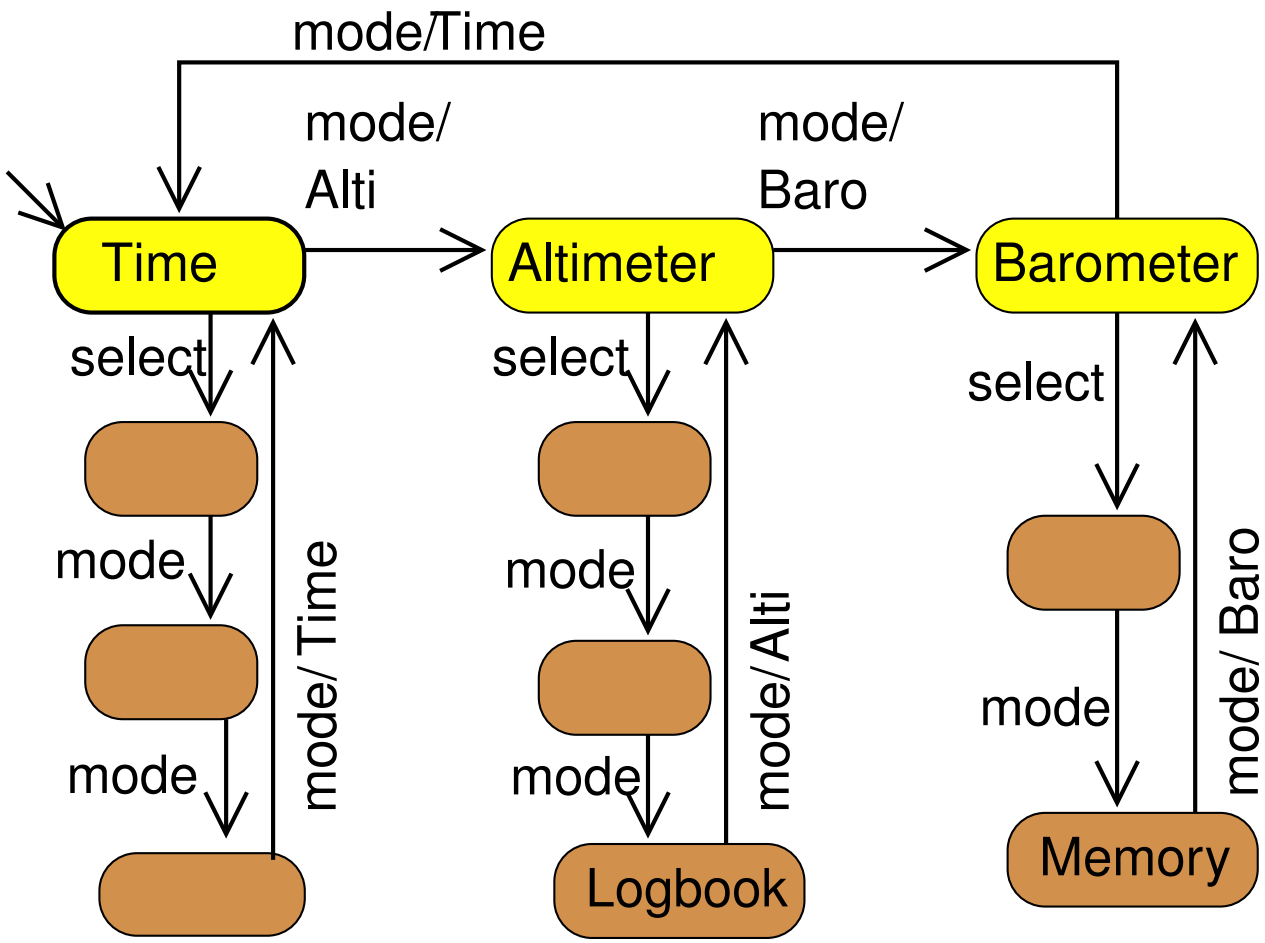
Joint Weaving

- Problem: aspect M written for altimax, not for altimax◁LB
- Idea: weave aspects jointly into the program
- Select join points for all aspects first, then apply advice

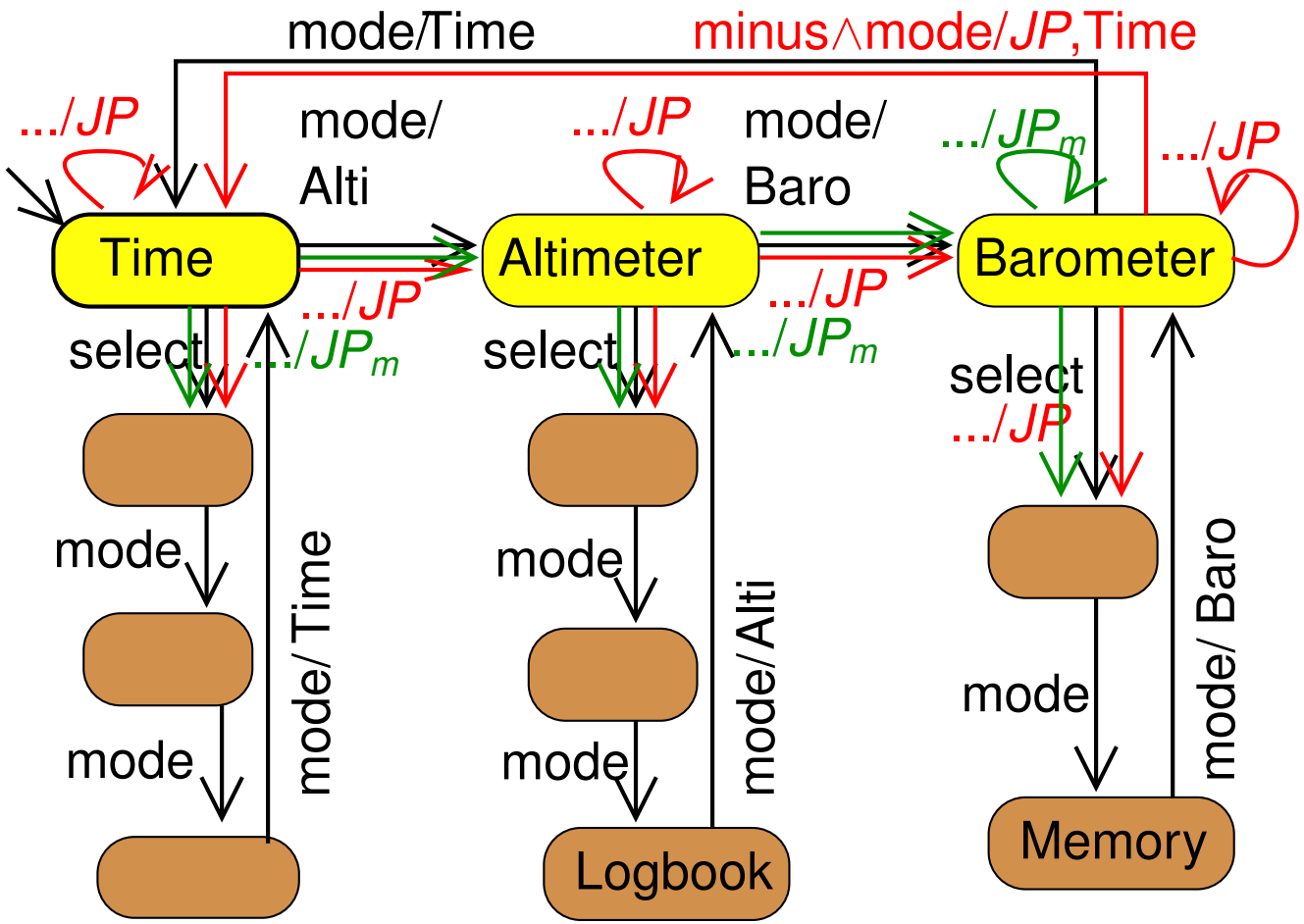
Joint Weaving: altimax◁(LB,M)

1. apply pointcuts and determine join point transitions
2. sequentially apply advice

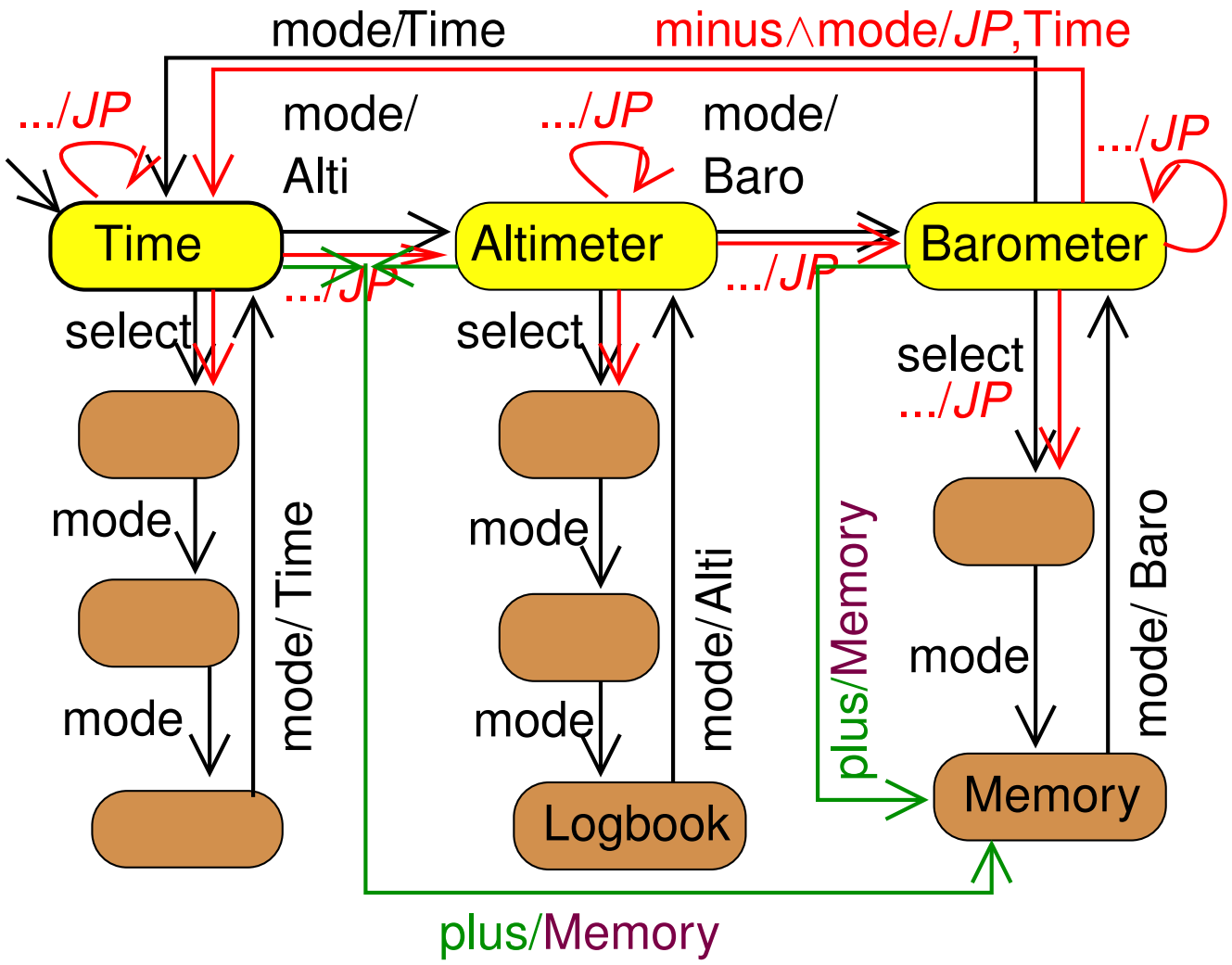
Application to the Example: altimax◁(LB,M)



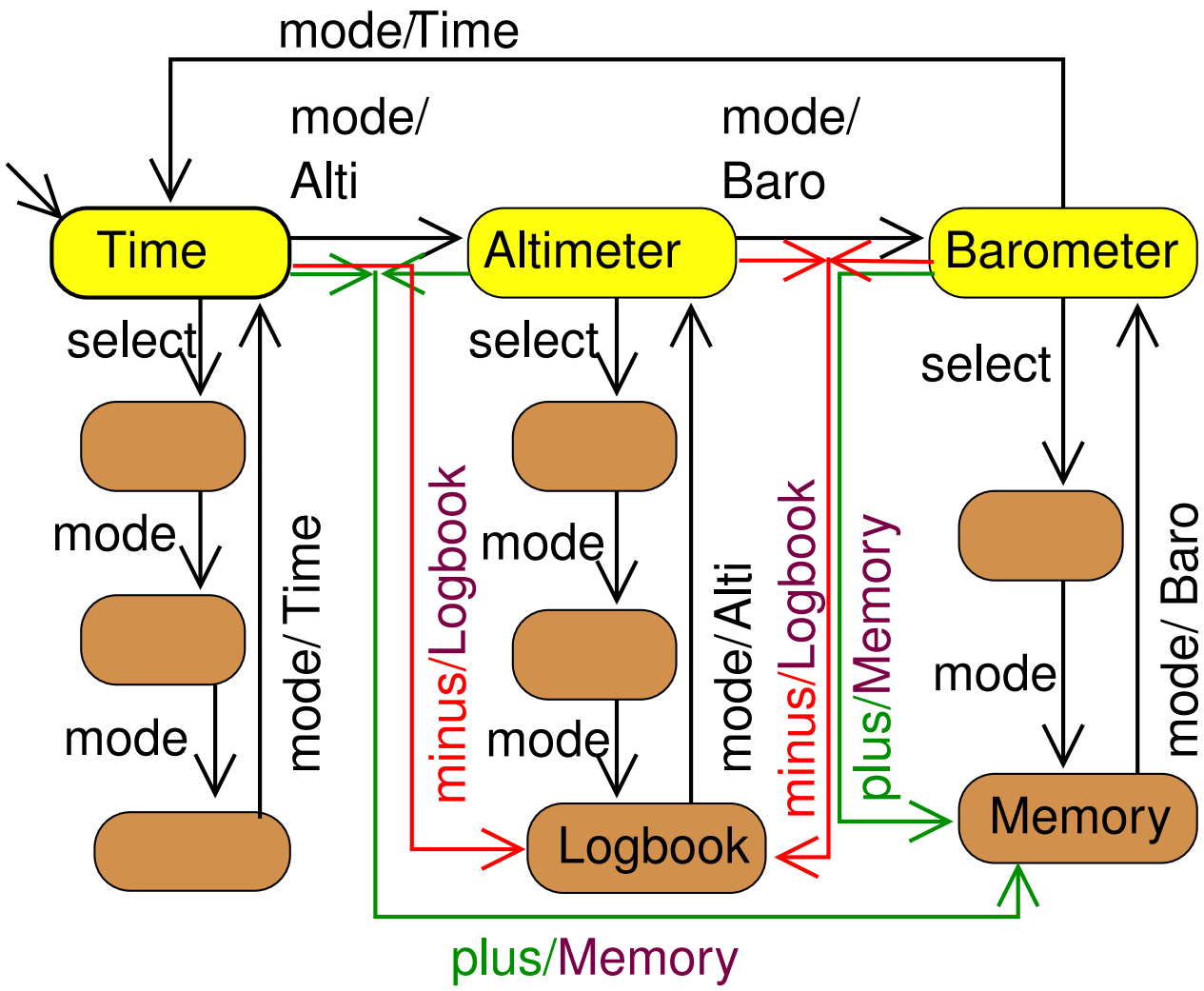
Application to the Example: altimax◁(LB,M)



Application to the Example: altimax◁(LB,M)



Application to the Example: altimax◁(LB,M)



Proving Non-Interference

- Is $\text{altimax}_{\triangleleft}(LB, M) \sim \text{altimax}_{\triangleleft}(M, LB)$?

Proving Non-Interference

- Is altimax$\triangleleft$(LB,M)$\sim$ altimax$\triangleleft$(M,LB)?
- Not always, because advice is still applied sequentially
- Jointly woven Larissa aspects still interfere,
  if they select the same join point transitions

Proving Non-Interference

- Is altimax$\triangleleft$(LB,M)$\sim$ altimax$\triangleleft$(M,LB)?
- Not always, because advice is still applied sequentially
- Jointly woven Larissa aspects still interfere,
  if they select the same join point transitions

Theorem for Jointly-Woven Aspects

- Noninterference of two aspects, for any base program:
  - if no transition selected by both aspects
    in product of pointcuts

Proving Non-Interference

- Is altimax$\triangleleft$(LB,M)$\sim$ altimax$\triangleleft$(M,LB)?
- Not always, because advice is still applied sequentially
- Jointly woven Larissa aspects still interfere,
  if they select the same join point transitions

Theorem for Jointly-Woven Aspects

- Noninterference of two aspects, for any base program:
  - if no transition selected by both aspects
    in product of pointcuts
- Noninterference of two aspects, for given base program P:
  - if no transition selected by both aspects
    in product of pointcuts and P

Design-by-Contract

- Originally introduced by Bertrand Meyer for object-oriented programming
- Contract: assumption $A \Rightarrow$ guarantee G

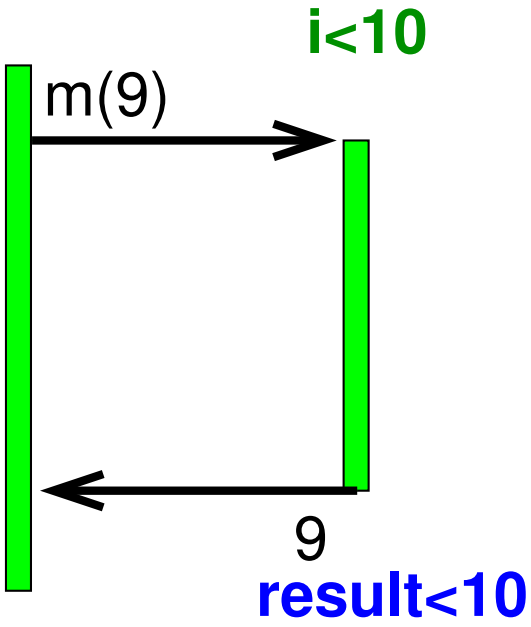
Design-by-Contract

- Originally introduced by Bertrand Meyer for object-oriented programming
- Contract: assumption $A \Rightarrow$ guarantee G
- Example (in Java):

```
class C{
  /* @ assume i < 10 */
  /* @ guarantee \result < 10 */
   int m(int i ){...}
}
```

Aspects Modify Contracts

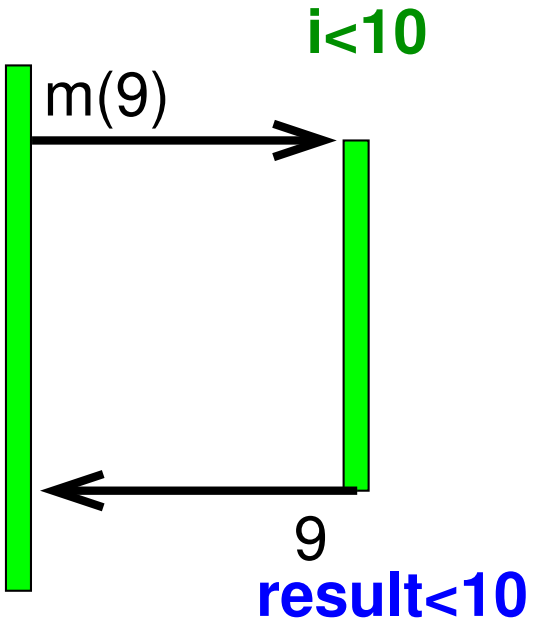
- Example call to m: i=9, returns 9



Aspects Modify Contracts

- Example call to m: i=9, returns 9
- Adding aspect to m:

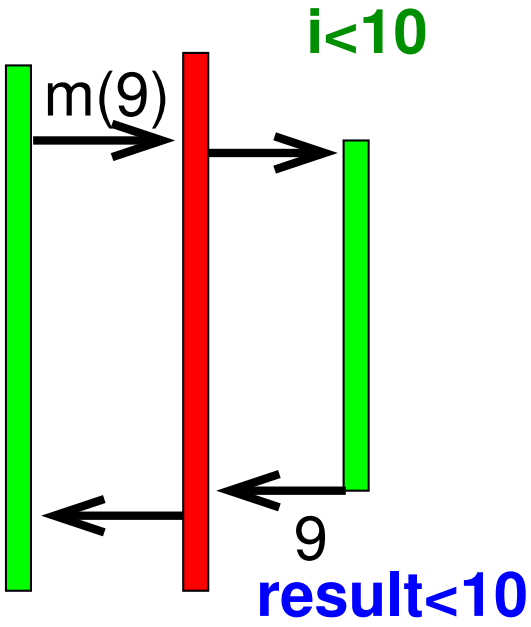
```
int around(int i): m(i){
    return 1 + proceed(i + 1);
}
```



Aspects Modify Contracts

- Example call to m: i=9, returns 9
- Adding aspect to m:

```
int around(int i): m(i){
    return 1 + proceed(i + 1);
}
```

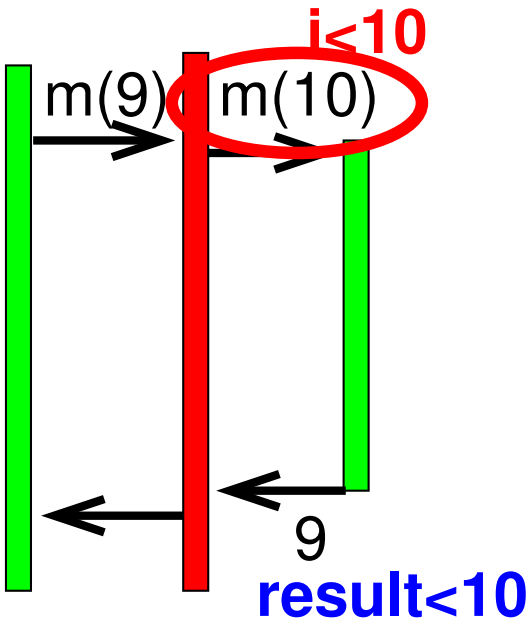


Aspects Modify Contracts

- Example call to m: i=9, returns 9
- Adding aspect to m:

```
int around(int i): m(i){
    return 1 + proceed(i + 1);
}
```

- Now: *A* violated

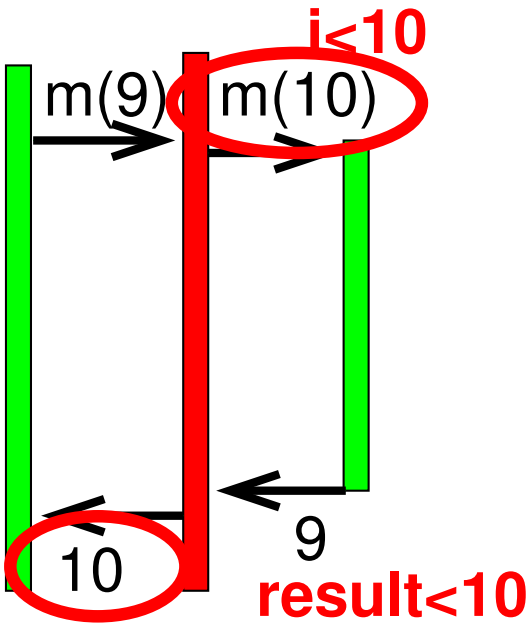


Aspects Modify Contracts

- Example call to m: i=9, returns 9
- Adding aspect to m:

```
int around(int i): m(i){
    return 1 + proceed(i + 1);
}
```

- Now: *A* violated, *G* violated



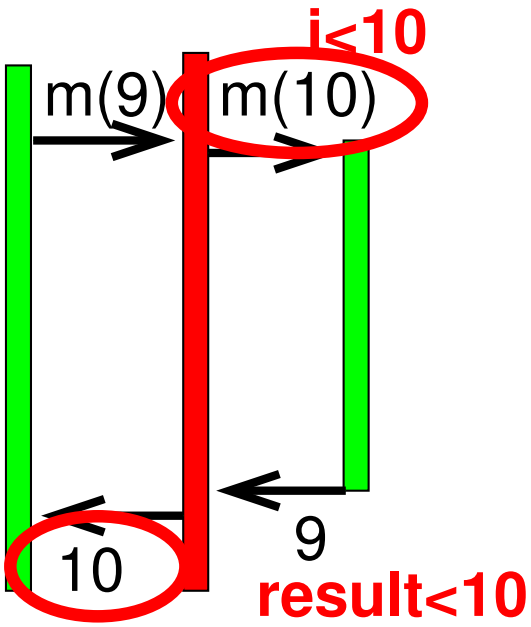
Aspects Modify Contracts

- Example call to m: i=9, returns 9
- Adding aspect to m:

```
int around(int i): m(i){
    return 1 + proceed(i + 1);
}
```

- Now: *A* violated, *G* violated
- In this case, a new contract for method with aspect can be derived:

```
/* @ assume i < 10 */
/* @ guarantee \result < 10 */
```



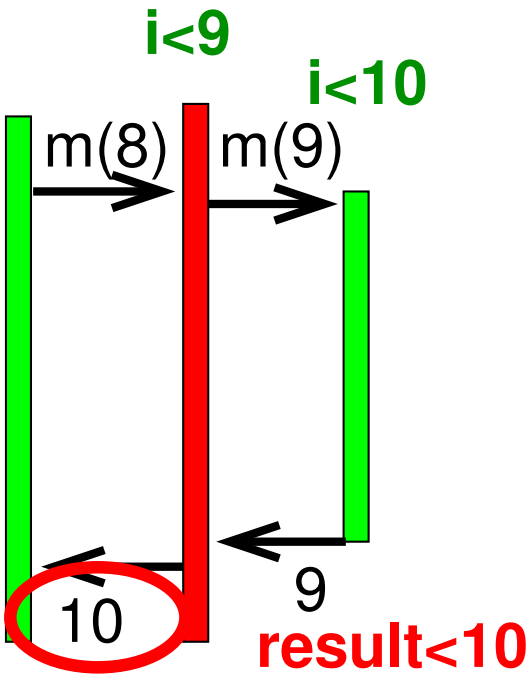
Aspects Modify Contracts

- Example call to m: i=9, returns 9
- Adding aspect to m:

```
int around(int i): m(i){
    return 1 + proceed(i + 1);
}
```

- Now: *A* violated, *G* violated
- In this case, a new contract for method with aspect can be derived:

```
/* @ assume i < 9 */
/* @ guarantee \result < 10 */
```



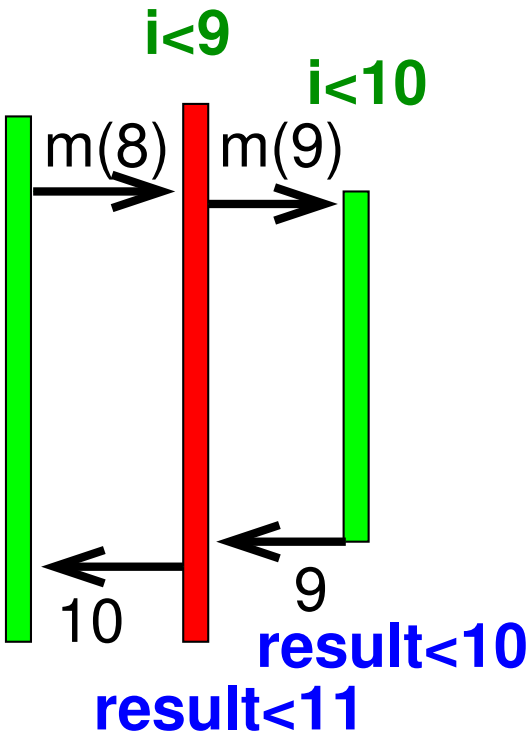
Aspects Modify Contracts

- Example call to m: i=9, returns 9
- Adding aspect to m:

```
int around(int i): m(i){
    return 1 + proceed(i + 1);
}
```

- Now: *A* violated, *G* violated
- In this case, a new contract for method with aspect can be derived:

```
/* @ assume i < 9 */
/* @ guarantee \result < 11 */
```



Aspects Modify Contracts

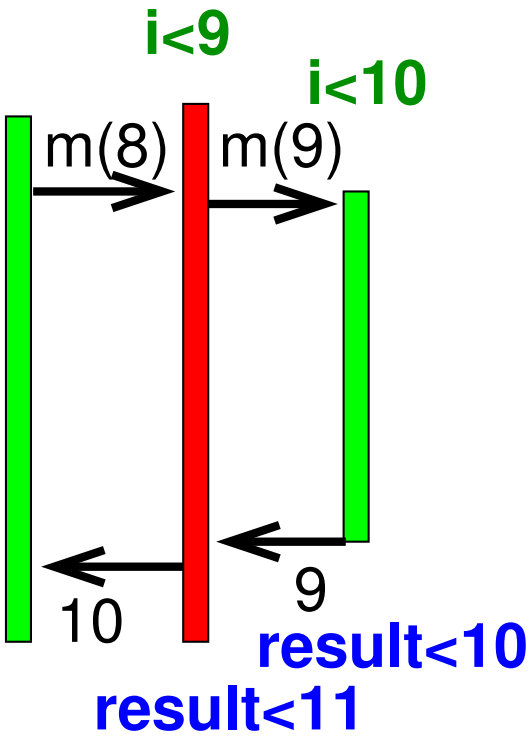
- Example call to m: i=9, returns 9
- Adding aspect to m:

```
int around(int i): m(i){
    return 1 + proceed(i + 1);
}
```

- Now: *A* violated, *G* violated
- In this case, a new contract for method with aspect can be derived:

```
/* @ assume i < 9 */
/* @ guarantee \result < 11 */
```

- Idea: derive new contracts automatically

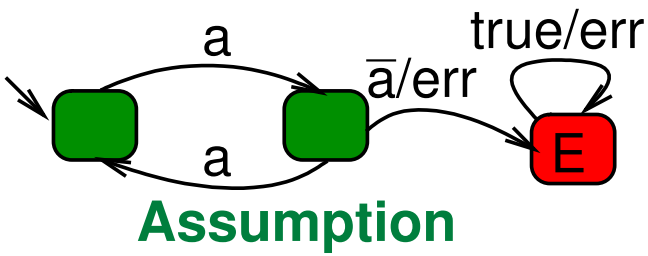


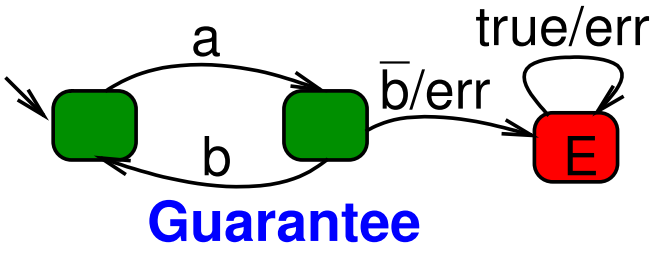
Contracts for Reactive Systems

- Assumption constrains inputs
- Guarantee constrains outputs
- Example with input a and output b:
  - Assumption: a always occurs in pairs
  - Guarantee: a is immediately followed by b

Contracts for Reactive Systems

- Assumption constrains inputs
- Guarantee constrains outputs
- Example with input a and output b:
  - Assumption: a always occurs in pairs
  - Guarantee: a is immediately followed by b
- Observers can express such properties
- Inputs are accepted until output err is emitted





Generating New Contracts

- Goal: apply *asp* to (A, G), and obtain (A', G'), such that

$$P \models (A, G) \Rightarrow P \triangleleft asp \models (A', G')$$

- Idea: Simulate the effect of the aspect on the program as far as possible on A and G
- Done for Argos and Larissa aspects
- Advantages of the approach:
  - determine effect of the aspect on program before it is written
  - allows modular verification

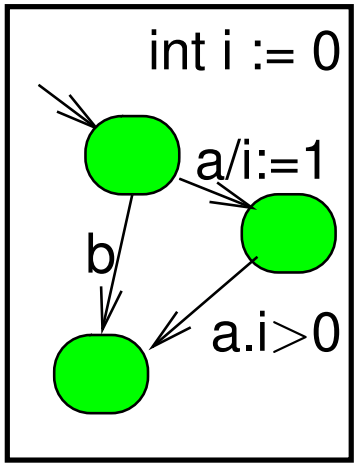
Outline

Contributions

- Contributions:
  - Identification of cross-cutting concerns in reactive systems
  - Larissa, an aspect language for Argos
    - formal definition and properties, preservation of equivalence
    - developed many examples and case studies
  - Formal analysis tools for Larissa
    - Non-Interference
    - Combination with Design-by-Contract
  - Compiler for Larissa
    - implements all language variants
    - handles large programs

Further Work

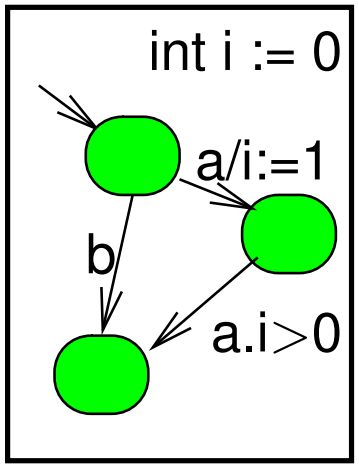
- Extension with variables
  - possible to respect encapsulation
- Extension to other synchronous languages
- Non-functional concerns in reactive contexts
  - Modeling of systems-on-a-chip
  - Modeling of sensor networks



Outline

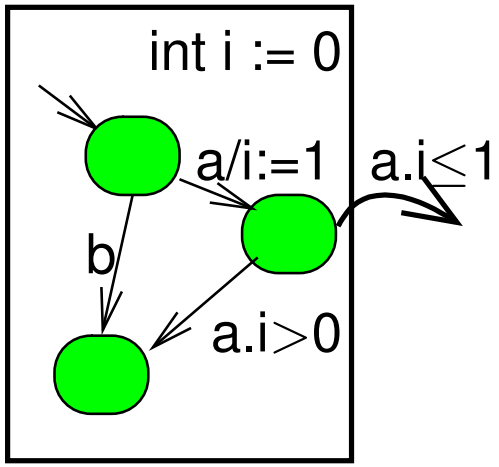
Extension with Variables

- Difficulty: respect encapsulation
- Internal integer variables:
  part of implementation



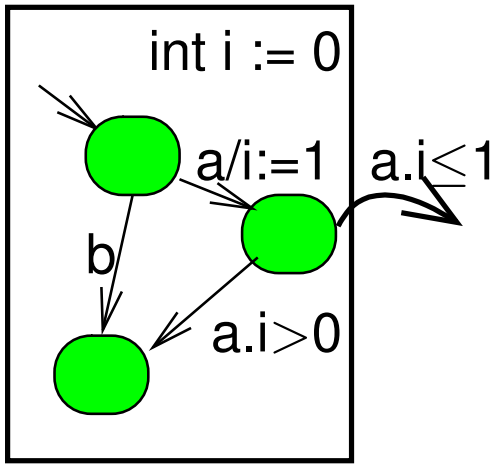
Extension with Variables

- Difficulty: respect encapsulation
- Internal integer variables:
  part of implementation



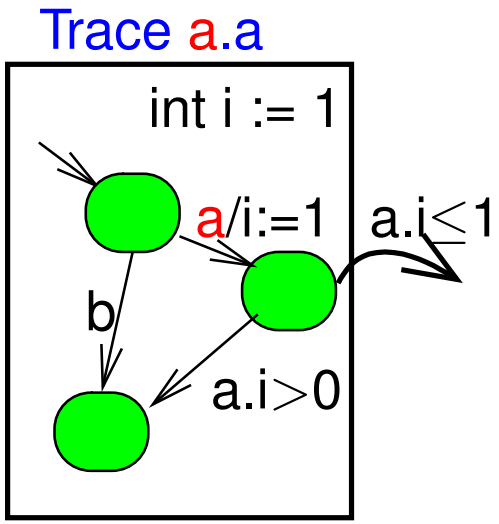
Extension with Variables

- Difficulty: respect encapsulation
- Internal integer variables:
  part of implementation
  - aspect must not change them directly
  - pointcut, advice program cannot use them, but can have their own
  - trace execution must set them



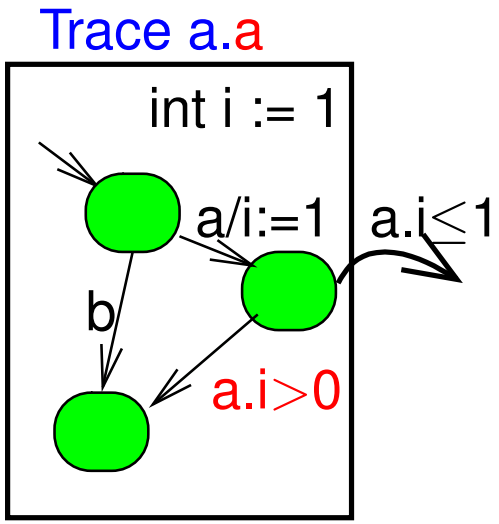
Extension with Variables

- Difficulty: respect encapsulation
- Internal integer variables:
  part of implementation
  - aspect must not change them directly
  - pointcut, advice program cannot use them, but can have their own
  - trace execution must set them



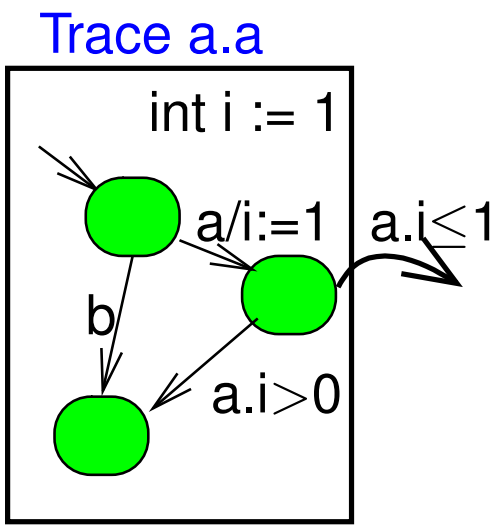
Extension with Variables

- Difficulty: respect encapsulation
- Internal integer variables:
  part of implementation
  - aspect must not change them directly
  - pointcut, advice program cannot use them, but can have their own
  - trace execution must set them



Extension with Variables

- Difficulty: respect encapsulation
- Internal integer variables:
  part of implementation
  - aspect must not change them directly
  - pointcut, advice program cannot use them, but can have their own
  - trace execution must set them
- Integer In/Outputs:
  - Aspect can modify outputs only if program cannot read them
  - i.e., o:=o+1 impossible



Aspect Languages for Other Synchronous Languages

- Synchronous languages have different styles:
  - Argos, Esterel: Imperative base + parallel composition
  - Lustre: purely parallel
- Adapt Larissa to other languages?
  - Pointcut: powerful, semantic, built-in everywhere
  - Advice: similar for Esterel (but trace automata specific)
  - Lustre: something different needed

Non-Functional Properties in Reactive Contexts

- Modeling and simulation of reactive systems
- Start by abstract functional model, add non-functional properties
- Non-functional properties often cross-cutting
- Identified two areas
  - Systems-on-a-chip: add timing information
  - Investigated in Quentin Meuniers Master's Thesis
  - Wireless sensor networks: energy consumption

Contract Weaving — Technical Overview

- Problem: aspects cannot be applied directly to observer automata

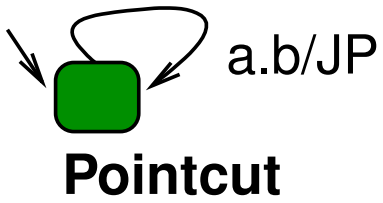
Contract Weaving — Technical Overview

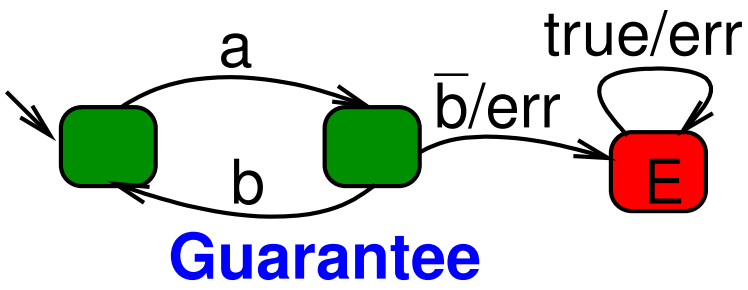
- Problem: aspects cannot be applied directly to observer automata
- Solution:
  - Transform observers into generator automata *nd*
  - Apply aspect to generators
  - Transform woven generators back to observers *obs*
  - Different for assumption and guarantee:
    - $A' = obs_A(nd_A(A) \triangleleft asp)$
    - $G' = obs_G(nd_G(G) \triangleleft asp)$
- Then,

$$P \models (A, G) \Rightarrow P \triangleleft asp \models (A', G')$$

Example – Guarantee Weaving

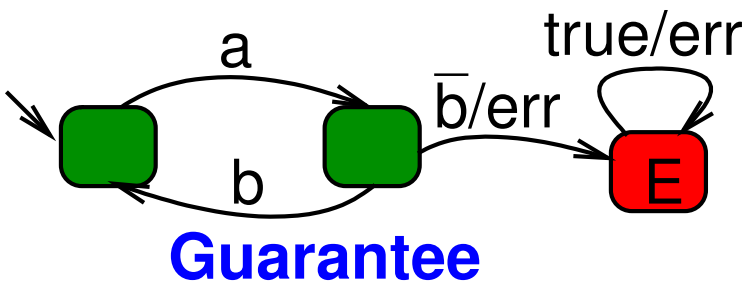
- Example aspect: advice output b, trace a

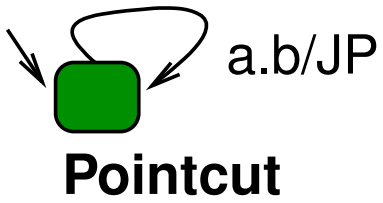


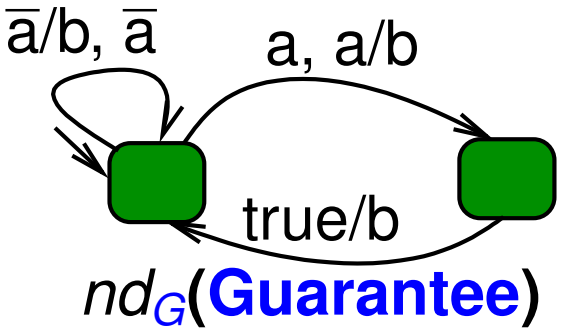


Example – Guarantee Weaving

- Example aspect: advice output b, trace a

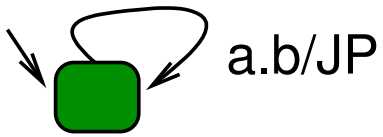




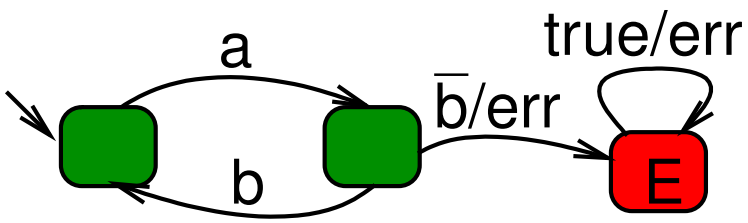


Example – Guarantee Weaving

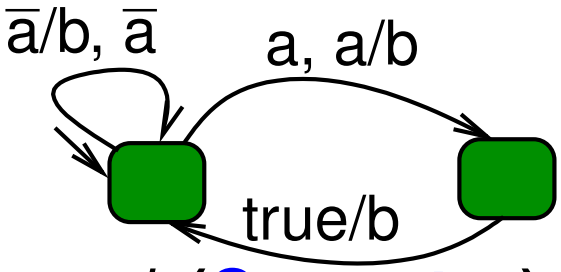
- Example aspect: advice output b, trace a



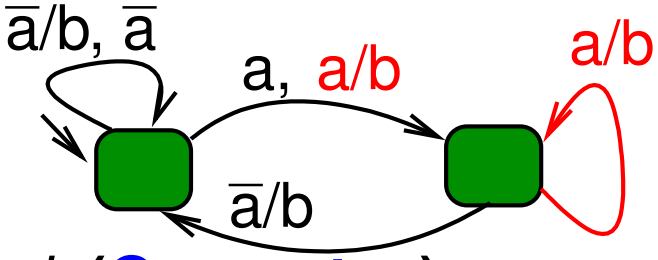
Pointcut



Guarantee



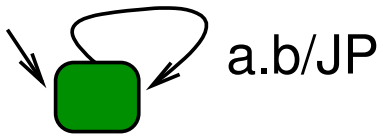
nd_G(**Guarantee**)



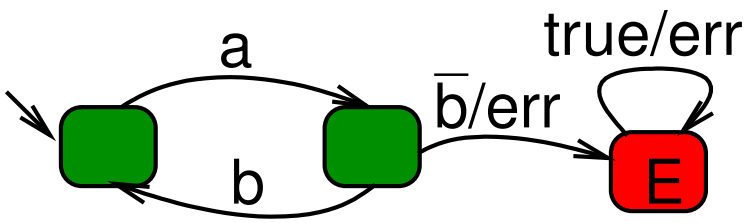
nd_G(**Guarantee**)$\triangleleft$*asp*

Example – Guarantee Weaving

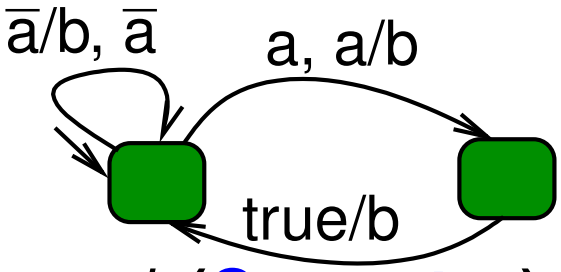
- Example aspect: advice output b, trace a



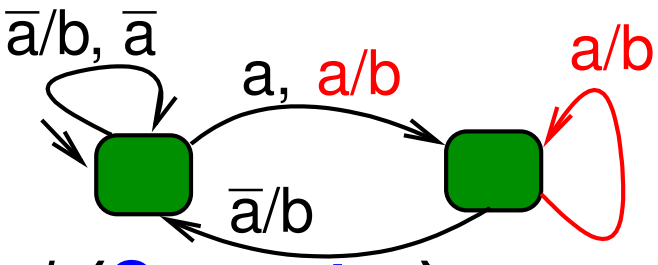
Pointcut



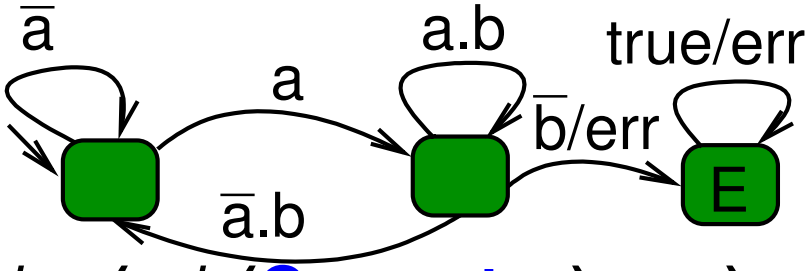
Guarantee



nd_G(**Guarantee**)



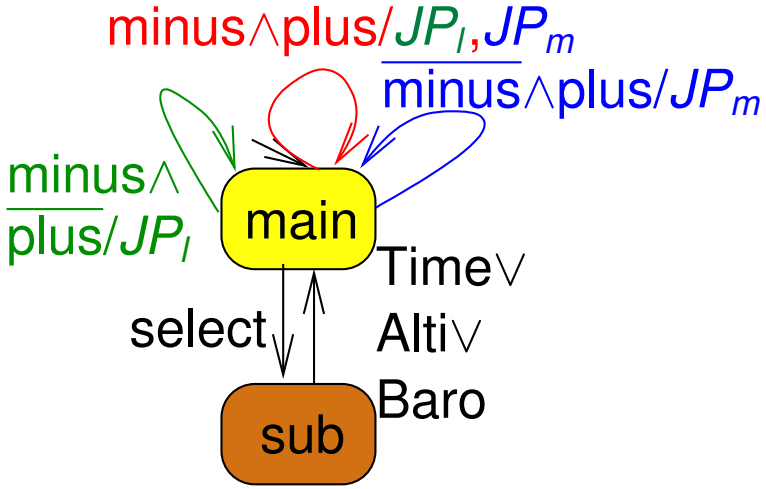
nd_G(**Guarantee**)$\triangleleft$*asp*



obs_G(nd_G(**Guarantee**)$\triangleleft$*asp*)

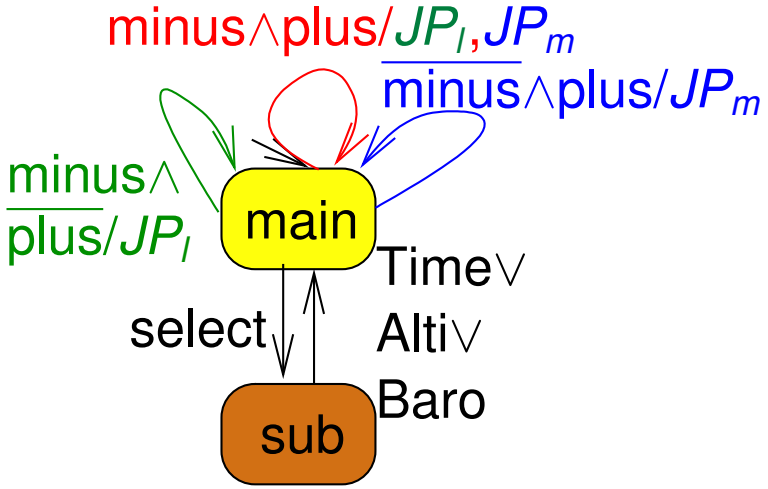
Interference: Shortcut Aspects

- Use first method: calculate product of two pointcuts



Interference: Shortcut Aspects

- Use first method: calculate product of two pointcuts



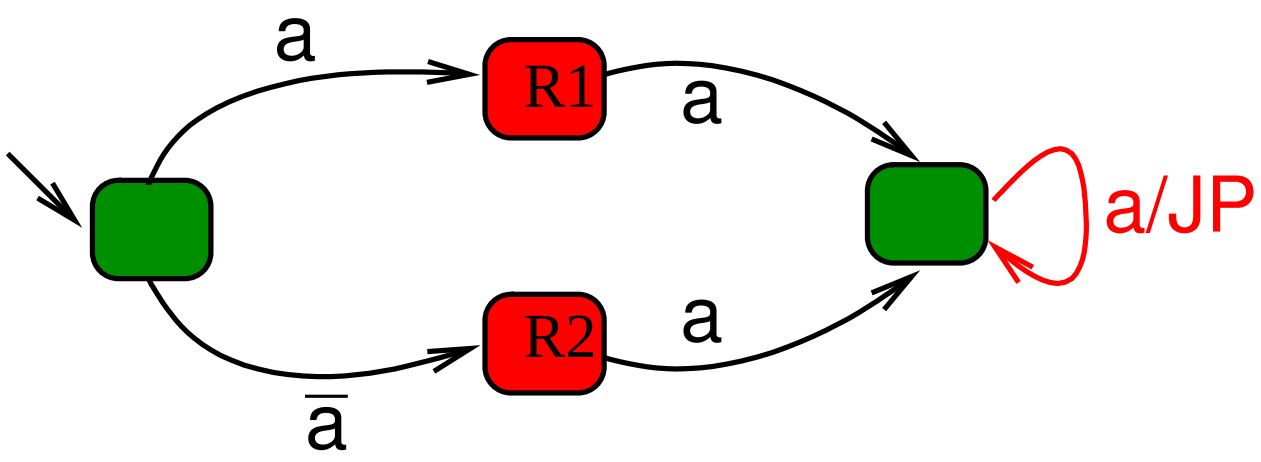
- Aspects interfere when both buttons are pressed at the same time in a main mode
- Product tells us exactly where aspects may interfere

Recovery Advice

- toInit advice: jumping to a fixed location
- toCurrent advice: jumping forward
- Missing: jumping backwards
- Specification with trace impossible: automaton not deterministic
- Different solution:
  - specify recovery states in base program
  - target state of advice transition: the last recovery state passed

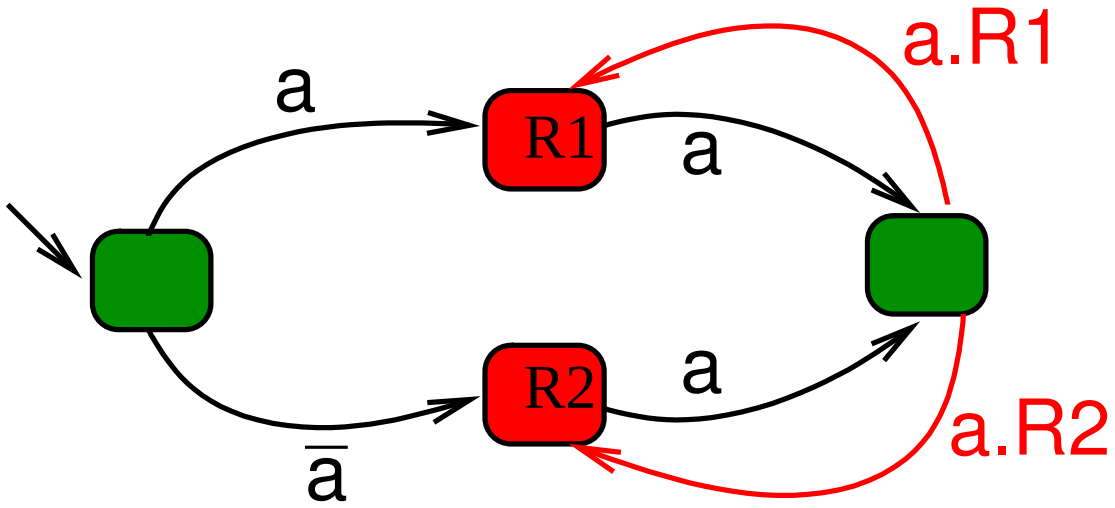
Example

- Example: R1, R2 recovery states (selected by an observer)
- Return to recovery state that was passed last



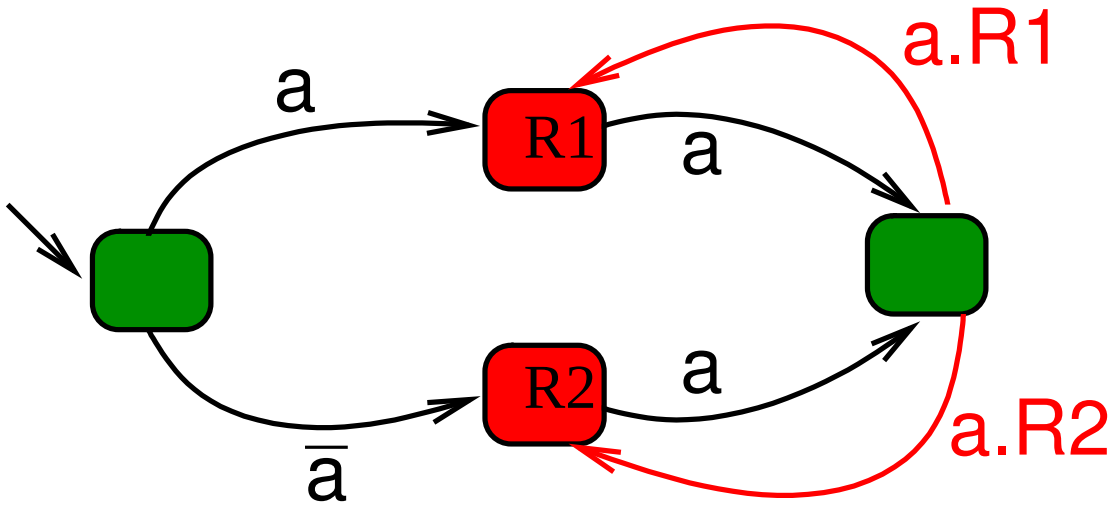
Example

- Example: R1, R2 recovery states (selected by an observer)
- Return to recovery state that was passed last



Example

- Example: R1, R2 recovery states (selected by an observer)
- Return to recovery state that was passed last



- Signals R1 and R2 decide which transition is taken
- Must be emitted by an Memory Automaton, run in parallel, which remembers which recovery state was passed last