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
1. Introduction
   - Aspect-Oriented Programming
   - Reactive Systems and Synchronous Languages
   - AOP for Reactive Systems

2. Larissa

3. Formal Analysis Tools

4. Conclusion and Further Work
Aspect-Oriented Programming

Example: figure editor
Aspect-Oriented Programming

- Example: figure editor
- Program: **modules** implement concerns

<table>
<thead>
<tr>
<th>Point</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>setX(int)</td>
<td>update()</td>
</tr>
<tr>
<td>setY(int)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>setP1(Point)</td>
</tr>
<tr>
<td>setP2(Point)</td>
</tr>
</tbody>
</table>
Aspect-Oriented Programming

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

<table>
<thead>
<tr>
<th>Point</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>setX(int)</td>
<td>update()</td>
</tr>
<tr>
<td>setY(int)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>setP1(Point)</td>
</tr>
<tr>
<td>setP2(Point)</td>
</tr>
</tbody>
</table>
Aspect-Oriented Programming

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

<table>
<thead>
<tr>
<th>Point</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>setX(int)</td>
<td>update()</td>
</tr>
<tr>
<td>setY(int)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>setP1(Point)</td>
<td></td>
</tr>
<tr>
<td>setP2(Point)</td>
<td></td>
</tr>
</tbody>
</table>
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
- 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

**Point**
- setX(int)
- setY(int)

**Line**
- setP1(Point)
- setP2(Point)

---

```java
// Example in Java and AspectJ

Point
  | setP1(P)
   |    |
   | setX(2)

Line
  | setP1(P)
   |    |
   | setX(2)
```

**AOP: Key Concepts**
- **Join points**: where aspects intervene
- **Pointcut**: select join points
- **Advice**: what aspect does

```java
aspect updateDisplay{
  pointcut stateChanged() :
  call(void Point.set*(..))
  && !cflowbelow(stateChanged())

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

**Conclusion on AspectJ**
- Based on lexical elements
  - Powerful constructs
Example in Java and AspectJ

AOP: Key Concepts

- **Join points**: where aspects intervene
- **Pointcut**: select join points
- **Advice**: what aspect does
Example in Java and AspectJ

```java
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

```java
Point
setX(int)
setY(int)

Line
setP1(Point)
setP2(Point)

setStateChanged()
setX(2)
update()
```

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

```java
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

```java
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

![Wristwatch diagram]
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

- Time
- Altimeter
- Barometer
- Logbook
- Memory

Mode/Time
Mode/Alti
Mode/Baro
Argos, a Synchronous Language

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

Time, Alti, Baro,...
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
1. Introduction

2. Larissa
   - Context and Requirements
   - The Language
   - Example

3. Formal Analysis Tools

4. Conclusion and Further Work
Argos Operators

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

\[(A \parallel B) \{ \text{mod2} \}\]

\[A \]

\[\text{a/mod2} \]

\[\text{a/mod4} \]

\[A \parallel B \]

\[\text{a} \]

\[\text{a} \]
Argos Operators

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

\[(A \parallel B) \setminus \{\mod 2\}\]
Argos Operators

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

\[(A \parallel B) \setminus \{\text{mod2}\}\]
Encapsulation

- Argos programs form expressions, e.g.
  \[((A\parallel B) \setminus \{a\}) \parallel C\]

- A, B and C automata or Argos expressions
Encapsulation

- Argos programs form expressions, e.g.
  \[ ((A || B) \setminus \{a\}) \parallel C \]
- \( A, B \) and \( C \) automata or Argos expressions
- Interface: inputs, outputs
Encapsulation

- Argos programs form expressions, e.g.
  \[ ((A \parallel B) \setminus \{a\}) \parallel C \]
- 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.

\[((A \parallel B) \setminus \{a\}) \parallel C\]

- 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

\[((A' \parallel B) \setminus \{a\}) \parallel C \sim ((A \parallel B) \setminus \{a\}) \parallel C\]
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
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
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
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
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

![Diagram showing pointcuts](image-url)
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
  ⇒ transition in program selected
Pointcuts

- Must select transitions
- Solution: observer automaton
  - inputs: inputs and outputs of observed program
  - one output JP
- Pointcut emits JP
  \[ \Rightarrow \text{transition in program selected} \]
Pointcuts

- Must select transitions
- **Solution:** observer automaton
  - inputs: inputs and outputs of observed program
  - one output **JP**
- Pointcut emits **JP**
  \[ \Rightarrow \text{ transition in program selected} \]
- Transitions identified statically by parallel product
**Difficulty:** specify one new target state
**Difficulty**: specify one new target state

**Solution**: execute **finite input trace**
- automaton deterministic, thus always identifies **one** state
**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

![Diagram](image-url)
**Advice**

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

![Diagram of a finite automaton with transitions labeled 'a' and 'b', leading to a target state labeled 'Aspect'.]
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.mode`, output `Logbook`
Example: Logbook Shortcut Aspect **LB**

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

![Diagram showing the pointcut of LB](image-url)
Example: Logbook Shortcut Aspect \textbf{LB}

- \textbf{Pointcut}: transitions in main modes where \texttt{minus} is true
- \textbf{Advice}: trace \texttt{mode.select.mode.mode}, output \texttt{Logbook}

```
main

select

sub

minus/JP

Time ∨ Alti ∨ Baro

select

.../JP

mode

mode/Time

mode

.../JP

mode/Baro

.../JP

mode/Alti

.../JP

mode

select

.../JP

mode

.../JP

mode

.../JP

minus ∧ mode/JP, Time

minus ∧ mode/JP, Time

select

.../JP

mode

mode

.../JP

mode/Baro

.../JP

mode/Alti

.../JP

mode

Memory

altimax

Pointcut: transitions in main modes where \texttt{minus} is true

Advice: trace \texttt{mode.select.mode.mode}, output \texttt{Logbook}
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**

```
nodes
- main
- sub
- Time
- Altimeter
- Barometer
- Memory
- Logbook

relationships
- `select` from Time to Altimeter
- `mode` from Altimeter to Barometer
- `mode` from Barometer to Memory
- `mode` from Logbook to Altimeter
- `mode` from Altimeter to Logbook
- `minus` from Logbook to Altimeter
- `mode` from Logbook to Barometer
- `mode` from Barometer to Logbook
```

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

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

```
Compass
select
mode/mainMode
mode

advice program

Time ➔ Altimeter ➔ Barometer ➔ Compass
```

```
mode/..
mode/...
mode/...
mode/...

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

1. Introduction

2. Larissa

3. Formal Analysis Tools
   - Aspect Interference
   - Aspects and Contracts

4. Conclusion and Further Work
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

- Do several aspects influence each other?
- When is $P \triangleleft A_1 \triangleleft A_2 \sim P \triangleleft A_2 \triangleleft A_1$?
Aspect Interaction

- Do several aspects influence each other?
- When is $P_{A1} A2 \sim P_{A2} A1$?

Example: Second Shortcut Aspect $M$

- Also use **plus** button as shortcut in the main modes
- Pressing **plus** goes to the Memory mode
Weaving the Second Shortcut Aspect

- Weave $M$ into $\text{altimax}\triangleleft\text{LB}$

Diagram:

- Main
- Sub
- Select
- Time $\lor$
- Alti $\lor$
- Baro
- Plus/JP
- Advice transitions added to Logbook mode

Diagram:

- Time
- Mode/Alti
- Logbook
- Plus/Memory
- Altimax $\triangleleft$ LB
- Minus/...
Weaving the Second Shortcut Aspect

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

Pointcut of $M$

altimax$\triangleleft$LB
Weaving the Second Shortcut Aspect

- Weave $M$ into $\text{altimax} \triangleleft \text{LB}$
- When pressing $\text{minus}$ in main mode:
  - $\text{altimax} \triangleleft \text{LB}$ goes to submode
  - pointcut stays in main mode

Pointcut of $M$
Weaving the Second Shortcut Aspect

- Weave $M$ into $\text{altimax}<\text{LB}$
- When pressing $\text{minus}$ in main mode:
  - $\text{altimax}<\text{LB}$ goes to submode
  - pointcut stays in main mode
- **Error**: Advice transitions added to Logbook mode

Pointcut of $M$

altimax$<\text{LB}<M$
Joint Weaving

- Problem: aspect $M$ written for $\text{altimax}$, not for $\text{altimax} \triangleleft \text{LB}$
- Idea: weave aspects jointly into the program
Joint Weaving

Problem: aspect $M$ written for $altimax$, not for $altimax \triangleleft LB$

Idea: weave aspects **jointly** into the program

Select join points for all aspects first, then apply advice

**Joint Weaving: $altimax \triangleleft (LB, M)$**

1. apply pointcuts and determine join point transitions
2. sequentially apply advice
Application to the Example: $\text{altimax} \triangleleft (\text{LB}, \text{M})$
Application to the Example: \texttt{altimax}(\texttt{LB},\texttt{M})
Application to the Example: \texttt{altimax} \llcorner (LB,M)
Application to the Example: $\text{altimax} \triangleleft (LB, M)$
Proving Non-Interference

Is $\text{altimax} \ll (\text{LB}, M) \sim \text{altimax} \ll (M, \text{LB})$?
Proving Non-Interference

Is $\text{altimax} \ll (\text{LB}, M) \sim \text{altimax} \ll (M, \text{LB})$?

- 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 $\text{altimax} \triangleright (\text{LB, } M) \sim \text{altimax} \triangleright (M, \text{LB})$?
- 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 $\text{altimax} \triangleleft (L_B, M) \sim \text{altimax} \triangleleft (M, L_B)$?
- 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 $A \implies$ guarantee $G$
Design-by-Contract

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

```java
class C{
    static {
        @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);
  }
  ```

Now:

- A violated
- G violated

In this case, a new contract for method with aspect can be derived:

```
/∗
assume i < ∗
/
/∗
guarantee \result < ∗
/∗
```

Idea: derive new contracts automatically

```
m(9)
result<10
9
```
Aspects Modify Contracts

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

```
Idea : derive new contracts automatically
```

9

m(9)

i<10

result<10

9
Aspects Modify Contracts

- Example call to m: i=9, returns 9
- Adding aspect to m:
  ```java
  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:
  ```java
  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:
  ```java
  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 \( m \):
  ```java
  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 \( \text{result} < 11 \) */
  ```
Aspects Modify Contracts

- Example call to m: i=9, returns 9
- Adding aspect to m:
  ```java
  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
1. Introduction
2. Larissa
3. Formal Analysis Tools
4. Conclusion and Further Work
   - Contributions
   - Further Work
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
Appendix

Outline

- More on Further Work
- More on Contract Weaving
- More on Interference
- More on Recovery Advice
Extension with Variables

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

\[
\begin{align*}
\text{int } i & := 0 \\
a/i:=1 \\
b \\
a.i & > 0
\end{align*}
\]
Extension with Variables

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

![Diagram with nodes and edges:
- int i := 0
- a/i:=1
- a.i<1
- a.i>0
- b
]
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

\[
\begin{align*}
\text{int } & i := 0 \\
\text{a.i} & := 1 \\
\text{a.i} & \leq 1 \\
\text{b} & \\
\text{a.i} & > 0
\end{align*}
\]
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

Diagram:
```
int i := 1
a.i:=1
a.i>0
b
```

Trace: `a.a`
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

```
int i := 1
```

Diagram:
- Trace `a.a`
- `int i := 1`
- `a.i<1`
- `a.i>0`
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

Trace a.a

```plaintext
int i := 1
a.i:=1  a.i<1
b
a.i>0
```

Appendix
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
Problem: aspects cannot be applied directly to observer automata
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 \( a \)

Diagram:
- Pointcut: \( \text{a.b/JP} \)
- Guarantee: \( b/\text{err} \rightarrow \text{true/err} \)
- \( \text{a} \rightarrow \text{b/err} \)
- \( \text{E} \)
Example aspect: advice output \( b \), trace \( a \)

\[ \text{Guarantee} \]

\[ \text{Pointcut} \]

\[ \text{nd}_G(\text{Guarantee}) \]
Example – Guarantee Weaving

Example aspect: advice output $b$, trace $a$

$\text{ Guarantee}$

$\text{Pointcut}$

$\text{nd}_G(\text{Guarantee}) \triangleleft \text{asp}$
Example – Guarantee Weaving

- Example aspect: advice output b, trace a

Example aspect: advice output b, trace a
Interference: Shortcut Aspects

- Use first method: calculate product of two pointcuts

$$\text{minus} \land \text{plus} / J_{P_i}, J_{P_m}$$

$$\text{minus} \land \text{plus} / J_{P_m}$$

**Diagram:**

- **main**
  - Time $\lor$
  - Alti $\lor$
  - Baro
  - select

- **sub**
  - $\text{minus} \land \text{plus} / J_{P_i}$
Interference: Shortcut Aspects

- Use first method: calculate product of two pointcuts

\[
\text{minus} \land \text{plus}/JP_i, JP_m \\
\text{minus} \land \text{plus}/JP_m
\]

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

Return to recovery state that was passed last
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