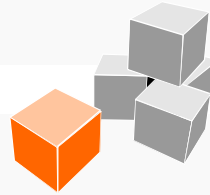
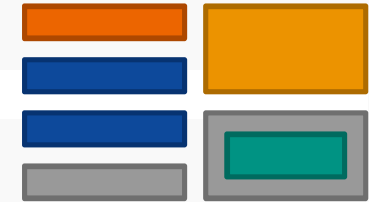




Twente Research and
Education on Software
Engineering, Universiteit
Twente



*Software
Technology
Group*
TU Darmstadt | FB Informatik



www.alia4j.org

Generic IDE Support for Dispatch-based Composition

Christoph Bockisch, and Andreas Sewe



Composition in this Talk



- Composition ...
 - ... facilitated by advanced languages
 - Multiple and Predicate Dispatching
 - Pointcut-Advice
 - Inter-Type Member Declarations
 - ... of programs written in different languages
- Subject of this talk ...
 - ... is **not**: new composition mechanism
 - ... **is**: support use of existing composition mechanisms

Current Limitations of IDE Support



- Languages for advanced composition
 - Often Research Prototypes
 - Extend base language
 - Implemented as code transformers
- Focus on language design
 - Few IDE tools
 - Little integration with base language tooling
 - No dedicated dynamic support

Current Limitations of IDE Support



- AspectJ Development Tools (AJDT) very advanced
 - Still has limitations
 - Much less support for similar languages
- Large number of languages overlapping in concepts
 - Should be possible to re-use tools supporting overlapping concepts

Examples of Limitations



- What are possible implementations to execute at call?

The screenshot displays four panels from an IDE:

- Code Editor:** Shows `DemoClass.java` with the following code:

```
1 package demo;
2
3
4 public class DemoClass {
5
6     public static void main(String[] args) {
7         Strategy strategy = StrategyFactory.createStrategy();
8         strategy.perform();
9     }
10
11 }
```
- Cross References:** Shows the `main(String[])` method with a call to `method-call(void demo.Strategy.perform())` and an `advised by` relationship to `Logging.before(): <anonymous pointcut>`.
- Hierarchy:** Shows the `Strategy` interface with two concrete implementations: `StrategyA` and `StrategyB`.
- Call Hierarchy:** Shows the `perform() : void - demo.Strat` method being called from the `main` method.

Examples of Limitations



- Why is call site affected by dispatch declaration?

```
DemoClass.java ✖
1 package demo;
2
3
4 public class DemoClass {
5
6     public static void main(String[] args) {
7         Strategy strategy = StrategyFactory.createStrategy();
8         strategy.perform();
9     }
10
11 }

Logging.aj ✖
1 package demo;
2
3
4 public aspect Logging {
5
6     before() : call(protected *.*(..)) ||
7         call(public * Strategy+.*()) {
8         System.out.println("Log");
9     }
10
11 }
```

Examples of Limitations



- Why is certain implementation executed at call at runtime?
 - Most advanced languages realized by code transformations
 - Must debug generated code



```
DemoClass.java
1 package demo;
2
3
4 public class DemoClass {
5
6     public static void main(String[] args) {
7         Strategy strategy = StrategyFactory.createStrategy();
8         strategy.perform();
9     }
10
11 }

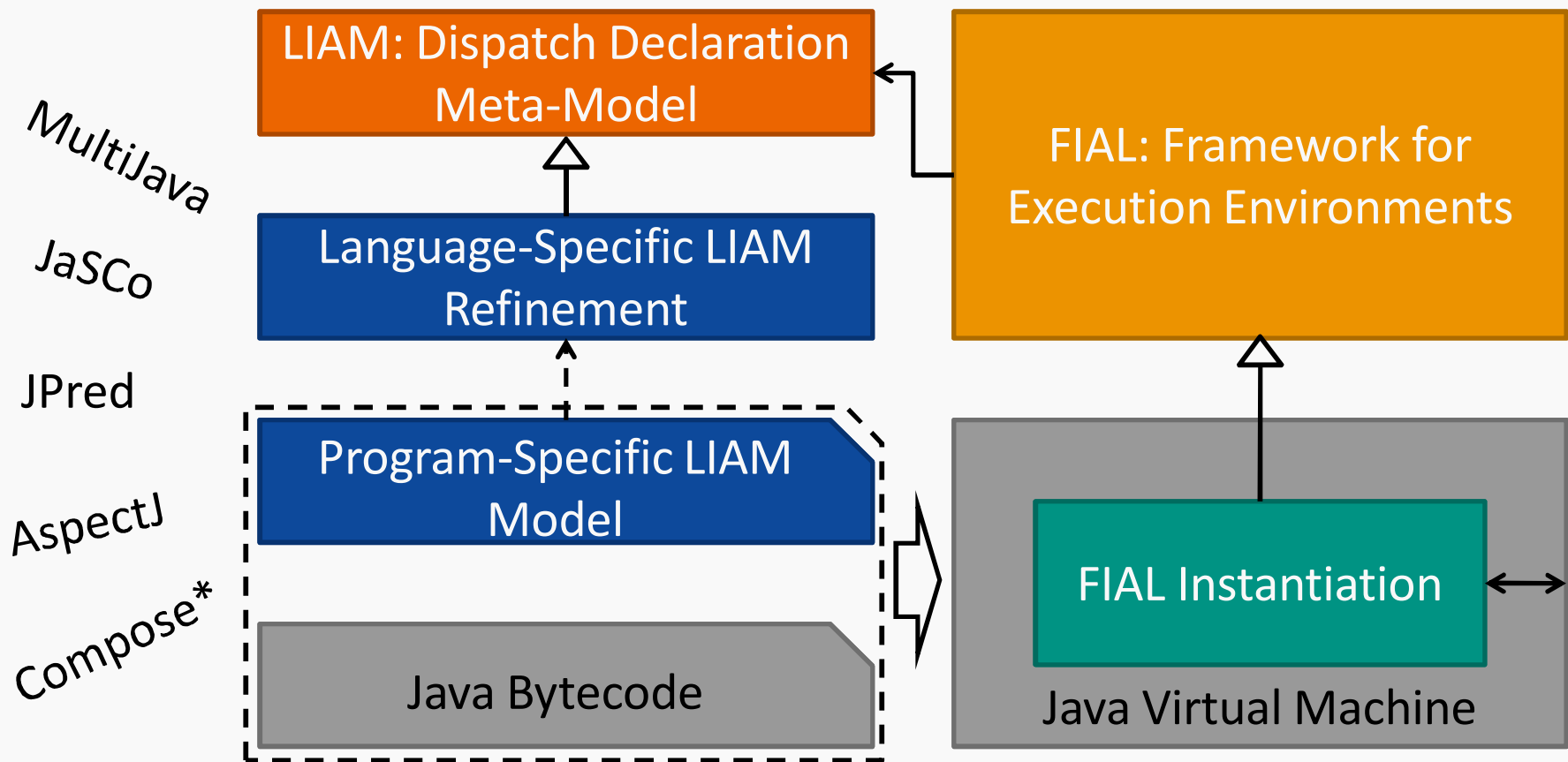
Logging.aj
1 package demo;
2
3
4 public aspect Logging {
5
6     before(StrategyA strategy) :
7         call(public * Strategy+.*()) &&
8         target(strategy) {
9         System.out.println("Log");
10    }
11
12 }
```

Goal and Approach of Research



- Requirements
 - Provide tools for languages with advanced dispatching
 - Support static and dynamic development tasks
 - Applicability to and re-usability by different languages
- Approach: based on Advanced-Dispatching Language-Implementation Architecture (ALIA)
 - Two declarative models: dispatch declaration and execution model
 - Lets languages share implementation of overlapping concepts
 - Modular implementation of work flows

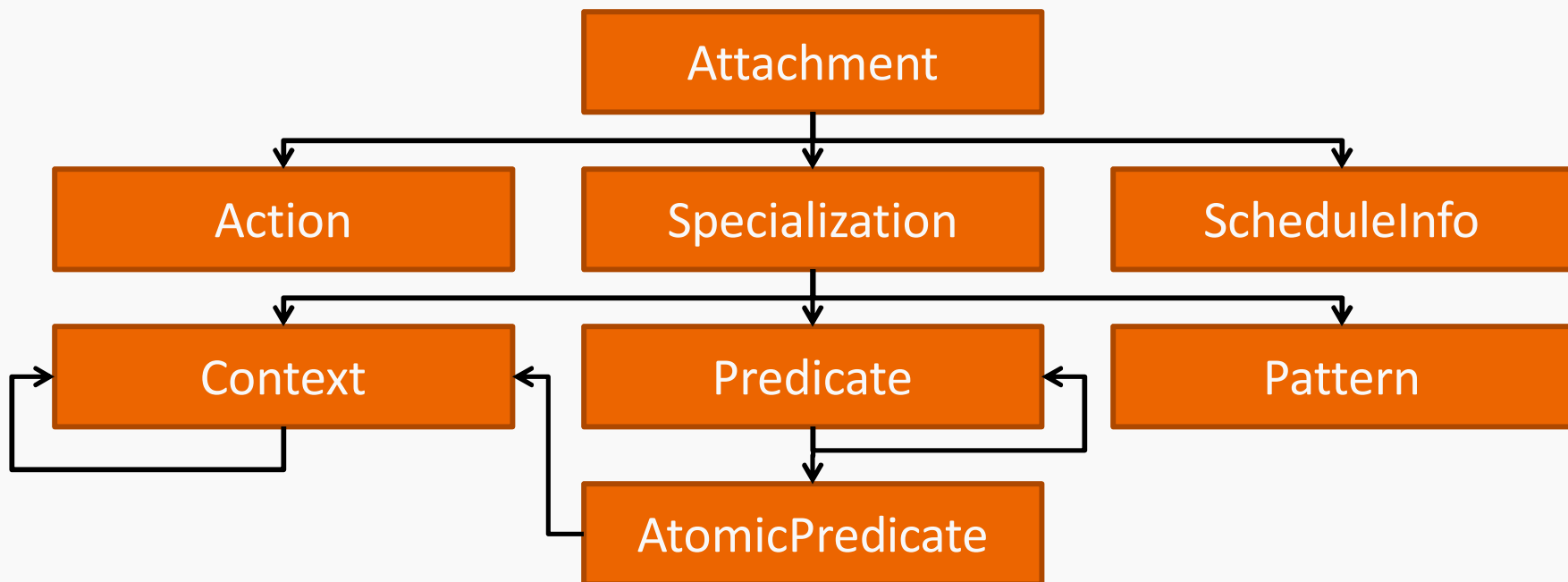
Advanced-Dispatching Language-Implementation Architecture (ALIA)



Dispatch Declaration Model



- Language support:
 - Implement meta-model refinements
 - Create model from (source) code

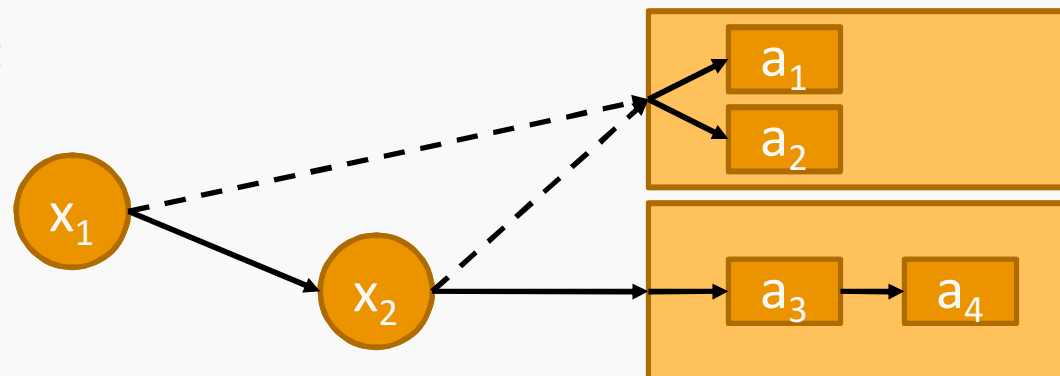


Dispatch Execution Model



- Dispatch function as binary decision diagram
 - Split: AtomicPredicate to evaluate
 - Sink: Actions to execute
- Actions to execute as tree
 - Supports nested actions like AspectJ's around
- Refers to elements of dispatch declaration

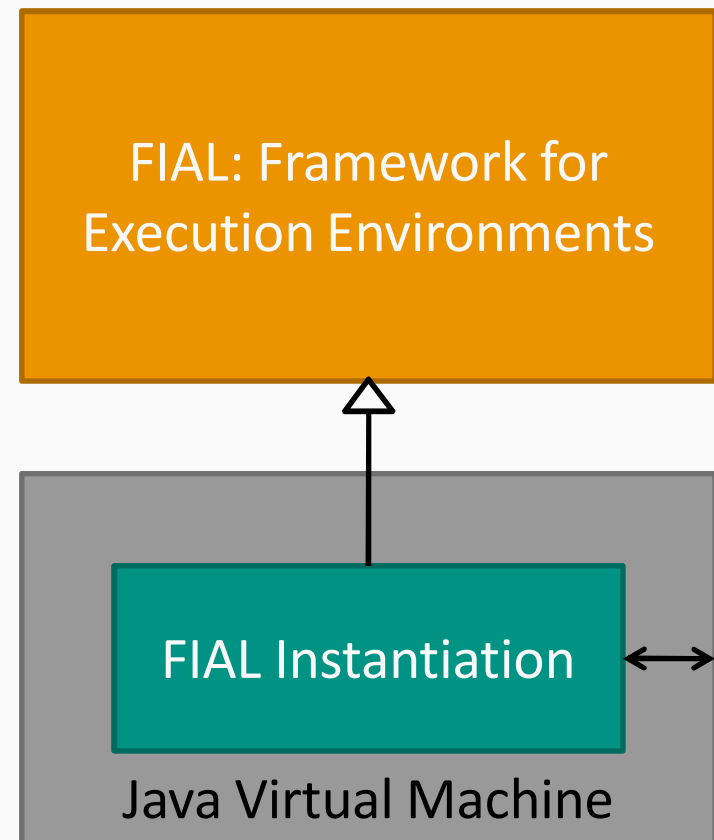
Example:



Modular Work Flows



- Multiple FIAL instantiations exist
 - Highly optimizing
 - **Interpreting**
- Work flows in FIAL
 - Partial evaluation of dispatch declarations
 - Pattern evaluation
- Work flows in FIAL instantiation
 - Execution model evaluation



General Idea



- LIAM (meta-model)
 - Declarative intermediate representation
 - Relatively close to source language concepts
 - Use for common program representation in IDE
 - Comparable to Java AST in Eclipse' Java Development Tools (JDT)
- FIAL / FIAL instantiations (work flows)
 - Visualize work flows and evaluation results
 - Execution environment needs to communicate with IDE (debugger)

General Idea



- What are possible implementations to execute at call?
 - Answered by dispatch execution model
- Why is call site affected by dispatch declaration?
 - Visualize pattern evaluations for call site
- Why is certain implementation executed at call at runtime?
 - Visualize evaluation path of binary decision diagram

Idea for Visual Debugging



The screenshot displays four panels from an IDE:

- Base.java**:

```
1  
2  
3 public class Base {  
4  
5     public static void main(String[] args) {  
6         A a = new B();  
7         C.m(a);  
8     }  
9  
10 }
```
- Logger.aj**:

```
public aspect Logger {  
    before() : call(* *.*(..)) && args(B) {  
        System.out.println("call with B");  
    }
```
- Dispatch Debug View**: A diagram showing the execution flow. A box labeled "ArgumentContext index='0'" points to a box "TypePredicate expected='B'". From "TypePredicate", arrows point to "C.m(A)" and "Logger.before1 C.m(A)".
- Variables**: A table showing the current state of variables.

Name	Value
1st arg	B (id=18)
B@d0a5d9	

Idea for Visual Debugging



```
Base.java
1
2
3 public class Base {
4
5     public static void main(String[] args) {
6         A a = new B();
7         C.m(a);
8     }
```

```
Logger.aj
public aspect Logger {
    before() : call(* *.*(..)) && args(B) {
        System.out.println("call with B");
    }
}
```

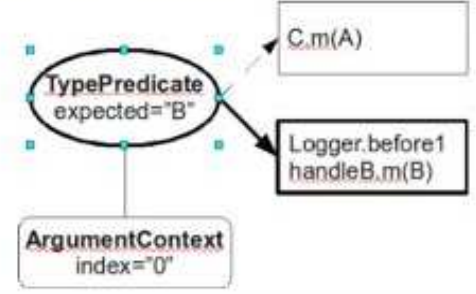
```
handleB.java
public void C.m(A&B b) {
}
```

Variables

Name	Value
1st arg	B (id=18)

B@d0a5d9

Since FIAL can execute attachments originating from multiple languages at once, the IDE can also support multi-language development.



Conclusions and Future Future Work



- Genericity:
 - Language independence and re-usability of concepts already shown
- Power:
 - LIAM and FIAL sufficient to answer relevant developer questions
- Additional research
 - Make user interface customizable for different languages
 - Use abstractions of source language instead of ALIA's