Aspects and Verification: Challenges and Opportunities

Shmuel Katz
Computer Science Department
The Technion
katz@cs.technion.ac.il
Topics

- What is an aspect?
- What are they good for in general?
- How can they help us test/debug/log?
- But are the aspects themselves correct?
  - How to specify
  - Kinds of aspects and properties
  - Approaches to verification
Aspects (and esp. AspectJ)

- Aspects: modular units that crosscut classes
- Aspects are defined by aspect declarations and may include
  - pointcut declarations: where to add/replace
  - advice declarations: what to add or do instead
  - Can introduce new methods, variables, code...
- **Weave** (=bind) aspect to different systems (but not entirely separated yet...)
Pointcuts

- A program element that identifies join points
  - Denotes a (possibly empty) set of join points
    - kind of join point
    - signature of join point
    - Can be dynamic (calls within a context, look at stack)

```
call(void Line.setP1(Point))
```

Denotes the set of method call join points with this signature
Advice

- Additional action to take at join points
  - Defined in terms of pointcuts
  - The code of a piece of advice runs at every join point picked out by its pointcut

```java
pointcut move() :
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point));
```

```java
after() returning : move() {
  < code here runs after completion of each join point denoted by move >
}
```
Advantages of aspects

- A system concern is treated in one place, and can be easily changed.
- Evolving requirements can be added easily with minimal changes to previous version.
- Configurable components become practical (“On demand computing”).
- Reuse of code that cuts across usual class hierarchy to augment system in many places.
Modularity for Cross-cutting

For distributed:
- Deadlock detection: is the system stuck?
- Monitoring: gathering information on messages
- Fault-tolerance: resending messages on new paths

For Object Oriented
- Monitoring and debugging
- Adding security: Encode/decode messages
- Preventing overflow: Catch and correct when needed
- Enforcing a scheduling policy

Analyzing QOS and Performance
The Opportunities

- Already used for logging and tracing values
- Can be used for evaluating tests
- Can be used to augment a system with debug `assert` statements when needed
- Good for annotating (=marking up) a system for input to analysis tools
  - Formal Methods (software model checking)
  - Simulation
  - White-box test generation
Challenges

- How do we know the aspect itself is correct?
- When is it applicable?
- What new properties does it add?
- What does it maintain from the old system?
- An aspect itself is not a program, and its application should be `light weight`
Aspects as Subjects of Investigation

- Syntax: how to express them?
- Classification: What types are there?
  - Spectative: only observes/records
  - Regulative: affects control/termination
  - Invasive: changes values of existing fields
- Specification: what do they add, to what?
- Correctness/validation: how do we know they do what is intended?
Terminology

- **Underlying or Original or Basic (system):** the system before an aspect has been woven
- **Aspect:** pointcut plus advice (where + what)
- **Augmented (system):** the result after weaving in an aspect
Ideal Goal: verifying aspects

- Show once and for all that:
  - For every possible underlying system satisfying Assumptions of the Aspect,
  - For any legal combination (weaving) of the aspect and the underlying system,
  - The New Functionality will be true for the augmented system, and
  - All previous desirable properties are still OK
The Problem: Impracticality

- Such a proof must be inductive
- No one really does inductive proofs for arbitrary software using existing tools
- Requires generalizations hard to express on every software architecture within a class, or every weaving of a certain type
- Expressing the specification itself can be hard
Overcoming the Problem: Divide and Conquer

- Cause no harm versus add desired properties
- Analyze just the aspect
  - For every possible weaving and classes of properties
  - For a specific weaving and given properties
- Analyze the augmented system — automatically after a manual one-time set-up
- Use static code analysis, restricted inductions, and model checking ---as needed
Do aspects applied to an original system cause harm?

- Assume the original system has a **specification** of its essential properties.
- Show that the aspects maintain those properties (but can change others).
- Ignore the properties added by the aspects—at least “Do No Harm”.
- Limits the obliviousness of the system to aspects applied over it; if “harm is caused”, at least be aware of it.
Possible Approaches

- Regression testing
- Static code type analysis
- Verification using induction
- Model checking

Aspect code analysis: consider only the aspect code, (a) for families of systems or (b) for one instance

Augmented code analysis: consider the combination of the original and the aspects
Why not regression testing?

- Aspects make many changes at many points and can redirect control and results
- Entire computation paths/methods/fields are not tested
- Inherently global, for augmented system, and can demand excessive resources

Previous tests are often insufficient/irrelevant
Static aspect code analysis: Example—spectative aspects

- If the binding of aspect code to a system is only through explicit parameters, can see that only aspect fields are modified, and original control is unaffected (=spectative)
- Use data-flow techniques (define-use pairs)
- Thrm: For any original system, properties only involving original fields, methods, are not harmed by applying a spectative aspect.
- But: New method exposing a hidden value could be even in a spectative aspect ...
Another Example: Regulative Aspects

- Can establish by code analysis that the aspect can gather information, OR restrict operations that were possible in the original
- Theorem: Safety properties are maintained, but Liveness may be violated
- Examples:
  - Access control (e.g., passwords) as an aspect
  - Restrict choices to guarantee fair scheduling
Deductive verification for aspect code: Invariant extension

- IF I is an invariant of the original system, and is inductive, we can just show that
  \[ \{I\} \ t \ \{I\} \]
  holds for each action t of the aspect code, without considering when t is applied, and conclude that I is an invariant of the entire augmented system.

Useful example of aspect code analysis for a particular application, using info on original.
Example of invariant extension for a particular instance

- \((x>y>0)\) is an invariant of some system
- An aspect has the form
  \(<\text{complex}> \rightarrow \text{double} (x,y)\)

Then check \(\{x>y>0\} \text{ double}(x,y) \{x>y>0\}\)
and conclude \((x>y>0)\) is an invariant of the entire augmented system
(Note: no need to analyze \(<\text{complex}>\) )
Using Aspect Validation for augmented system analysis

For situations where original system has been proven correct for its specification using software model checking (e.g., Bandera)

- Reprove for augmented system without new manual setup (just push a button…)
- Reuse the specification and annotations, given as verification aspects
- Treats all new paths/methods…
- In many cases uses the same abstractions
On Aspect Validation

- Show each application of an aspect over a system is correct: “no harm” + new properties
- Still formal verification, but for each instance
- Key idea: set-up is manual, but then the proof for each instance is automatic
- Proves that applications so far are correct
- First used for Compiler Validation [Pnueli, Strichman, …]
Key ideas of Aspect Validation

- Use an existing software model checking tool
- Define collections of aspects, with specifications
- Use aspects themselves to express the annotations to systems needed for various model checking tasks (recall “opportunities”)
- Manual set-up is done once, then a sequence of automatically generated tasks are done each time the collection of aspects is woven into a basic system.
What is model checking?

- Given a **finite** representation of a **model** (a program), and an **assertion** about execution paths in temporal logic, check whether the assertion holds for every possible execution path (even infinite ones!) and thus is a **property** of the model.

- Generate compact representations, use clever algorithms to check, restrict assertion language, use **abstractions** and **reductions** to get smaller models, …
Software model checking

- Tool that allows annotating (Java) code, abstracting domains, expressing properties to be checked
- Bandera (or others) generate input to existing tools like SMV, Spin, ...
- For proper abstractions, success means the checked property holds for every execution
- Often ends with a counter-example
- Can fail due to state explosion, giving no info
- Algorithmic (except for finding abstractions)
Verification Aspects

Annotations to be added to Applications of Aspects over Original Systems

- For each Application Aspect, build 2 VA’s:
  - Asm: Assumptions of the Application
  - Res: Desired results of the Application

- Contain new fields, predicates, directives...for the application aspect.

- For each Original system, need another VA:
  - Spec: specification of the Original system
The Validation process

- **Correctness of Original**: Apply Spec to Original, and activate model checker (done earlier)

- **Original is appropriate**: Apply Asm to Original, activate model checker

- **Apply Application over Original giving A+B**,
  - **No harm**: Apply Spec to A+B, activate model checker
  - **Achieves result**: Apply Res to A+B, activate model checker
When will this work?

- The bindings for the application are the same as those needed for the verification aspects.
- The abstraction for the spec. of the original still works for the augmented.
- One generic abstraction for the new aspect properties works for many bindings to different systems, and can be remembered.
- Otherwise, the application is not automatic.
Validation gives a practical path to routine application

- Only expert needs to write annotations (once)
- Practical limitations:
  - Tools have arbitrary restrictions
  - Need abstractions
- Counter-examples can find bugs
- The key: full modularization of the VA’s allows automatic application
Some Interesting Goals

- Identifying classes of aspects + systems + properties appropriate for static type analysis or inductive proofs or model checking only for the aspect
- Analyzing when abstractions and reductions that were effective for model checking the original system and specification work for the augmented system
- Discovering generic abstractions and reductions that can be reused to model check the augmented system for new aspect properties
- Analyzing interference / cooperation among aspects
Conclusions

- Aspects are interesting
  - New kind of modularity (cross-cutting)
  - Potential for “on-demand” adaptation
  - Relevant for all stages of software development
- Formal Methods for software are interesting
  - Elegant applications of mathematics (logic)
  - Software crisis in reliability, expensive debugging
  - Tools are finally becoming practical
- Their combination has especially interesting questions and is potentially useful and practical
Sources

- S. Katz, Diagnosis of Harmful Aspects Using Regression Verification, FOAL workshop in AOSD 2004