Interactions and AOSD: 
Past, current and future

F. Sanen

Report CW 525, September 2008

Department of Computer Science, K.U.Leuven

Abstract

Interaction problems between different modules (be it aspects or software entities of another nature) are a key challenge when studying and creating advanced software engineering techniques that support modularity and separation of concerns. Interaction problems are manifest at all stages of the software engineering process and it is important to start addressing such problems as early as possible. This session will survey recent initiatives towards the management of feature interaction problems in an AOSD development cycle. Related work will be covered. A number of open research challenges will be discussed. This CW report contains the course materials of the course “Feature interaction and AOSD: Status and Future” given at the 3rd European summer school on Aspect-Oriented Software Development from July 21-25, 2008 in Darmstadt, Germany.
Interactions and AOSD: Past, current and future

outline

• Problem statement
• A historic perspective
• State of the art
• Towards the future
Problem statement

• Composing aspects automatically can introduce various types of errors
• Undesirable effects
  – Aspect execution order can matter
  – Aspect can require the presence of others
  – Aspects can be incompatible with each other
  – ...

Outline

• Problem statement
• A historic perspective
• State of the art
• Towards the future
Example setting

- Client-server system
- Client requests various resources from the server
- Client configuration can involve
  - Cache aspect
  - Authentication aspect
  - Logging aspect
  - Encryption aspect

Execution Order

- If network delays occur...
  - Cache aspect is needed next to the already present authentication aspect
  - Authentication aspect must be executed first
Dependency

• If every authentication attempt should be logged...
  – Logging aspect needs to be executed each time authentication aspect is executed
  – Authentication aspect requires logging aspect

Mutex

• If application needs to operate in secure mode...
  – Encryption feature on the cache could be considered to be introduced
  – Encryption and caching aspect are incompatible because of security issues with caching encrypted objects
Conflict

• If both encryption and logging aspect are required...
  – Do we want encrypted logs or log information that was not encrypted?
  – We need to make a trade-off

Things can go wrong
Interactions are not all bad

Aspect interactions

• Definition
  – Different definitions exist
    • Syntactic (structure based)
    • Semantic (behaviour based)
  – Popular:
    situation where one aspect that works correctly in isolation does not work correctly anymore when it is composed with other aspects
Aspect interactions

• Aspect can be involved in arbitrary number of interactions
• Different kinds of interactions
  – Dependency
  – Conflict
  – Choice
  – Mutex
  – Assistance
  – Execution order

Interaction classification

• Dependency
  – When an aspect explicitly needs another one
  – No problem if dependency is instantiated
  – E.g. authorization and authentication

• Conflict
  – Semantic interference
  – Has to be solved by mediation, trade-off, ...
  – E.g. confidentiality and audit
Interaction classification

• Choice
  – When you have two equivalent concerns
  – No need to have the aspects realizing both concerns because of same net effect
  – E.g. different authentication mechanisms

• Mutex
  – Realizing an aspect prohibits realizing another one
  – No mediation possible
  – E.g. strong timing constraint vs. heavy security

Interaction classification

• Assistance
  – When an aspect influences the correct working of another one positively
  – Extended functionalities become possible and/or extra support is offered
  – E.g. caching encrypted data, location context enables richer authorization, ...

• Execution Order
  – E.g. authentication must be performed before caching
Aspect interactions

• Classifications
  – Useful for understanding both the problem space and the solution space
  – Different classifications exist
    • Clifton et al. (2002)
    • Douence et al. (2002, 2004)
    • Rinard et al. (2004)
    • Katz et al. (2006)
    • Sanen et al. (2007)
    • Munoz et al. (2008)

Clifton et al. (2002)

• Spectators and assistants: enabling modular aspect-oriented reasoning
  – Focus: divide aspects into two sorts automatically to support modular reasoning
  – Spectators: cannot change behavior of modules
    • Stated by the aspect itself (ideally can be statically verified)
  – Assistants: not limited in this way
    • Stated by the module by naming allowed assistants
Douence et al. (2002, 2004)

• Composition, reuse and interaction analysis of stateful aspects
  – Focus: aspects sharing the same join point can result in undeterministic weaving
  – Notion of strong independence (versus independence w.r.t. a program)
  – Example: encryption and logging

Rinard et al. (2004)

• Classification system and analysis for AOP programs
  – Focus: direct and indirect interactions between advice and (base) methods
  – Direct categories
    • Augmentation, narrowing, replacement, combination
  – Indirect categories
    • Orthogonal, independent, observation, actuation, interference
Katz et al. (2006)

• Aspect categories and classes of temporal properties
  – Focus: syntactically identify aspect categories to simplify correctness proofs
  – Notion of aspect semantics to be specified
    • Assumptions on underlying systems (preconditions)
    • Required to be true afterwards (postconditions)

Katz et al. (2006)

• Aspect categories and classes of temporal properties
  – Categories (also defined in terms of state machines)
    • Spectative (only change aspect-local state)
    • Regulative (affect control flow)
    • Invasive and weakly invasive (change state)
  – Temporal properties
    • Safety (always hold, cfr. invariants)
    • Liveness (guaranteed to hold eventually)
    • Existence (can hold in at least one execution path)
Sanen et al. (2007)

• Managing concern interactions in middleware
  – Focus: explicitly represent knowledge about aspect interactions
  – Five categories
    • (Orthogonal)
    • Assistance
    • Dependency
    • Conflict
    • Mutual exclusion
    • Choice

<table>
<thead>
<tr>
<th>Feature interaction</th>
<th>Possible Combinations</th>
<th>( \varepsilon(A,B) ) Interaction Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orthogonal</td>
<td>{A, B}, {A}, {B}</td>
<td>( \varepsilon(A) + \varepsilon(B) )</td>
</tr>
<tr>
<td>Complements</td>
<td>{A, B}, {A}, {B}</td>
<td>( \varepsilon(A, B) ) &gt; ( \frac{\varepsilon(A) + \varepsilon(B)}{2} )</td>
</tr>
<tr>
<td>Dependent</td>
<td>{A, B}, {A}</td>
<td>( \varepsilon(A, B) ) &gt; 0, ( \varepsilon(A) ) &lt; 0</td>
</tr>
<tr>
<td>Conflicts</td>
<td>{A}, {B}</td>
<td>( \varepsilon(A, B) ) &lt; 0</td>
</tr>
<tr>
<td>Prevents</td>
<td>{A}, {B}</td>
<td>( \varepsilon(A) )</td>
</tr>
<tr>
<td>Equivalent</td>
<td>{A, B}, {A}, {B}</td>
<td>( \varepsilon(A) \rightarrow \varepsilon(B) ) ( \varepsilon(A,B) \rightarrow \varepsilon(B) )</td>
</tr>
</tbody>
</table>

Munoz et al. (2008)

• Improving maintenance in AOP through an interaction specification framework
  – Identify different patterns according to which AspectJ aspects can be invasive
    • Advice invasiveness categories
      – Augmentation, conditional replacement, replacement
      – Multiple, crossing, write, read, argument passing
    • Aspect invasiveness categories
      – Hierarchy, field addition and operation addition
Outline

• Problem statement
• A historic perspective
• State of the art
• Towards the future
Back in time

• Feature interactions heavily studied within telecommunications domain
  – More than a decade of research results
  – Feature interaction workshop (FIW) series
  – Surveys exist for the telco domain
    • Keck et al. (1998)
      – “The feature and survey interaction problem in telecommunications systems: a survey”
    • Calder et al (2003)
      – “Feature interaction: a critical review and a considered forecast”
    • ...

Let’s start with an example...

Call Waiting
• Suspend a call when a second call is answered

Call Forwarding
• Forward a call when phone already is in use
And another one...

Call blocking
- Block a call from X when X is trying to reach you

Call Forwarding
- Forward a call when phone already is in use

From the telco world

- Three major research areas and their results
  - Software engineering
    - Includes specification, development, testing and deployment of services
      → adapt service creation to help remove feature interactions
  - Formal methods
    - Formal description, modelling and reasoning techniques
      → Primarily used for detecting service level interactions (i.e. independent of any implementation)
  - Online techniques
    - Techniques applied at runtime
      → Allow interactions to occur at runtime; detect them and recover from them
From the telco world

• Software engineering
  – Advantages
    • Can help directly via particular methods, notations and techniques
    • Also helps indirectly because of effectiveness and rigouresness

From the telco world

• Software engineering
  – 2 major approaches
    • Focussed techniques: particular technique for elimination of interactions
      – Lots of work exists
    • Process models: redefining complete development process
      – Few approaches have been proposed
Software engineering research

From the telco world

• Formal methods
  – Advantages
    • Force assumptions and contexts to be made explicit
      *Incorrect or ambiguous assumptions are often the source*
    • Raise the level of documentation
      *System (its formal model) can be interrogated*
    • Make it easier to define classes of interactions in a more abstract way
    • Automated analysis and reasoning techniques are usually applicable
From the telco world

- Formal methods
  - 3 major approaches
    - Properties
      - Abstract properties are defined, usually in logic
      - Interactions then expressed in terms of that logic, e.g. as inconsistency or unsatisfiability
    - Behaviour
      - Behavioural description of features and basic service, usually in terms of automata and transition systems
      - Interactions then expressed in variety of generic ways, e.g. deadlocks or non-determinism
    - Both

Formal methods research

<table>
<thead>
<tr>
<th>Reference</th>
<th>Behaviour Language</th>
<th>Interaction Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTL</td>
<td>SDL</td>
<td>none</td>
</tr>
<tr>
<td>LOTOS</td>
<td>LOTOS</td>
<td>simulation</td>
</tr>
<tr>
<td>µ-calculus</td>
<td>LOTOS</td>
<td>restricted model-checking</td>
</tr>
<tr>
<td>TLA</td>
<td>LOTOS</td>
<td>goal oriented trace execution</td>
</tr>
<tr>
<td>MSC</td>
<td>SDL</td>
<td>simulation</td>
</tr>
<tr>
<td>CTL</td>
<td>LOTOS</td>
<td>symbolic transition checking</td>
</tr>
<tr>
<td>Symbolic transition predicates</td>
<td>P-EBF</td>
<td>symbolic transition checking</td>
</tr>
<tr>
<td>CTL</td>
<td>SMV</td>
<td>restricted model-checking</td>
</tr>
<tr>
<td>Lustre</td>
<td>linear past temporal logic</td>
<td>random tests</td>
</tr>
<tr>
<td>LTL</td>
<td>Promela</td>
<td>full model-checking</td>
</tr>
<tr>
<td>Finite State Automata</td>
<td>language difference</td>
<td></td>
</tr>
</tbody>
</table>

References:
- Gibson, Petry [24]
- Bouquet, Coudreuse, Rahier, Zimmer [36]
From the telco world

• Online techniques
  – Advantages
    • Applied on an active system (versus formal methods that operate on a model of the real system)
    • Support for quick time-to-market for new features
    • More future-proof since they are part of the system and not only the development process
    • Increasing multi-vendor market removes global knowledge about services in the network
    • Interactions can be detected and resolved without tampering legacy systems

From the telco world

• Online techniques
  – Major approaches
    • Location of control
      – Feature manager based (centralised)
      – Negotation based (decentralised)
    • Collection of information
      – A priori information: data collected at design time
      – Isolated online environment (at runtime, but not in live system)
      – At runtime
### Online techniques research

<table>
<thead>
<tr>
<th>Reference</th>
<th>Control</th>
<th>Information acquired at</th>
<th>Experimental Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morley, Magill [91]</td>
<td>Feature Manager</td>
<td>run time</td>
<td>case study, testbed</td>
</tr>
<tr>
<td>Hornsey, Singh [60]</td>
<td>Feature Manager</td>
<td>a-priori</td>
<td></td>
</tr>
<tr>
<td>Casa [21]</td>
<td>Feature Manager</td>
<td>a-priori</td>
<td></td>
</tr>
<tr>
<td>Fritschle [42]</td>
<td>Feature Manager</td>
<td>a-priori</td>
<td></td>
</tr>
<tr>
<td>Reiff [105]</td>
<td>Feature Manager</td>
<td>run time, general rules</td>
<td>case study</td>
</tr>
<tr>
<td>Tsang, Magill [112,113]</td>
<td>Feature Manager</td>
<td>captive environment</td>
<td>case study, testbed</td>
</tr>
<tr>
<td>Aggeoun, Cardoso [8]</td>
<td>Feature Manager</td>
<td>captive environment</td>
<td></td>
</tr>
<tr>
<td>Velthuijsen [125], Graf, Seth, Velthuijsen [21]</td>
<td>Negotiating Agents</td>
<td>a-priori</td>
<td>case study, industrial</td>
</tr>
<tr>
<td>Ender, Aanoy, Elamvaram, Quevenel, Gray, Monkowski [20]</td>
<td>Negotiating Agents</td>
<td>a-priori</td>
<td></td>
</tr>
<tr>
<td>Amer, Karmouch, Gray, Monkowski [4]</td>
<td>Negotiating Agents</td>
<td>a-priori</td>
<td>case study, industrial</td>
</tr>
</tbody>
</table>

### Lessons learned

- Online techniques
- Software engineering
- Formal methods
Lessons learned

😊 Problem understanding advanced greatly
😊 Telco industry encourages use of formal methods
😊 Feature interaction contests finding unexpected interactions

😊 Case studies have rarely been performed
😊 Strong concentration on POTS
😊 Software processes are mainly academic

Further in time

• Also other domains
  – Policy-based systems
    • E.g. Lupu et al. (1999)
  – Email
    • E.g. Hall et al. (2000)
  – Thermo control
    • E.g. Diaz Pace et al. (2000)
  – Multimedia:
    • E.g. Blair et al. (2001)
  – Middleware
    • E.g. Liu et al. (2005), Sanen et al. (2007)
  – ...
Outline

• Problem statement
• A historic perspective
• State of the art
• Towards the future
State of the art

• Software development lifecycle as a structure
• Classify existing works against two dimensions
  – Research area
    • Software engineering (SE)
    • Formal methods (FM)
    • Online techniques (OT)
  – Focus
    • Detection
    • Resolution
    • Managing

State of the art

• Requirements engineering
• Architecture and design
• Language support
• Middleware
• Full lifecycle support
Requirements engineering

• FODA
  – Feature-Oriented Domain Analysis
• NFR
  – Non-Functional Requirements framework
• AORE
  – Aspect-Oriented Requirements Engineering
• KAOS
  – Knowledge Acquisation in autOmated Specification

FODA

• Feature-Oriented Domain Analysis
  – Popular in software product lines
  – Feature model
    • Hierarchically arranged set of features
    • Modeling commonality and variability
    • Relationships between parent feature and its child features
      – And, alternative, or, mandatory, optional
    • Most current methodologies organize features in a tree, called feature diagram (FD)
      – So, FD is a graphical representation of a feature model
      – Each FD has a straightforward iterative tree grammar representation

Kang et al., Feature-Oriented Domain Analysis, 1990
FODA

• Interactions
  or “feature relations”
  – Requires
  – Excludes

• Tool support
  – pure::variants (http://www.pure-systems.com)
  – GEARs tool (http://www.biglever.com)
  – Captain Feature
    (https://sourceforge.net/projects/captainfeature)

Kang et al., Feature-Oriented Domain Analysis, 1990

NFR

• Non-Functional Requirements framework
  – Offering a structure for representing and
    recording the design and reasoning process of
    NFRs in softgoal interdependency graphs (SIGs)
  – Offering catalogues of knowledge about NFRs and
    design knowledge: NFR type, method, correlations

• SIG records the developer’s consideration of
  softgoals (main requirements) and their
  interdependencies

Chung et al., Non-Functional Requirements in Software Engineering, 2000
NFR

• Interactions or “softgoal interdependencies”
  – Explicit
    • SOME-
      – BREAK (--) 
      – HURT (-)
    • UNKNOWN (?)
    • SOME+
      – HELP (+)
      – MAKE (++)
  – Implicit
    • Detected positive contribution
    • Detected negative contribution

Chung et al., Non-Functional Requirements in Software Engineering, 2000

AORE

• Aspect-Oriented Requirements Engineering
  – Composition of crosscutting concerns, including their mutual influences and trade-offs
  – Requirements description language (RDL)
    • Close to natural language
  – Compositions can be specified both on the syntax of the natural language and on the semantics of the requirements
AORE

• Interactions or “trade-offs”
  – Compositions are formally expressed using base, constraint and outcome operators

\[
\text{LoggingComposition} = \forall r, s. \{ \text{Concern}(s) = \text{"AccessControl"} \} \rightarrow \\
( \exists k. \text{Meets}(\text{Time}(s), k) \land (\text{Meets}(k, \text{Time}(r)) \land \\
\text{pre}(s) = \text{In}(\text{"user"}, \text{"system"}) \land \neg \text{Authenticated}(\text{"user"}, \text{"system"}) \land \\
\text{post}(s) = \text{In}(\text{"user"}, \text{"system"}) \land \text{Authenticated}(\text{"user"}, \text{"system"}) \land \\
\text{Finishes}(\text{pre}(s), \text{Time}(s)) \land \text{Meets}(\text{Time}(s), \text{post}(s)) \\
\]  

  – Methods for revealing potential conflicts
    • Examine the logical conjunction of different compositions
    • Comparing time structure

AOSD-Europe, Analysis and Design Lab, 2008

KAOS

• Knowledge Acquisition in autOmated Specification
  – Goal-driven requirements acquisition
    • Represent each goal as a temporal logic rule
    • Decompose each goal into a set of subgoals by making use of refinement patterns
    • Also obstacles (negated goals) can be expressed
    • Assign each of the refined goals to specific object/operation such that the final system will meet the original requirements

• Formal techniques and heuristics are proposed for detecting conflicts

• Conflict classification into 9 categories
  – Conflict: logical inconsistency + minimality condition
  – Divergence: conflict + boundary condition
    • E.g. “User using resource as long as needed” & “user will free resource within some deadline”

Van Lamsweerde et al., Managing conflicts in goal-driven requirements engineering, 1998.
KAOS

• Interactions or “conflicts and divergences”
  – Detecting
    • Deriving boundary conditions by backward chaining
    • Using divergency patterns, e.g. achieve vs. avoid goals
    • Divergence identification heuristics
  – Resolving
    • Assertion transformation: operators which create, delete or modify goal assertions to avoid boundary conditions, restore goals, etc.
    • Object transformation: operators which create, delete or modify object types to refine objects or agents

Van Lamsweerde et al., Managing conflicts in goal-driven requirements engineering, 1998.

Requirements engineering

• FODA
  – Feature-Oriented Domain Analysis
• NFR
  – Non-Functional Requirements framework
• AORE
  – Aspect-Oriented Requirements Engineering
• KAOS
  – Knowledge Acquisation in automated Specification
Architecture and design

• De Fraine et al. (2008)

• Monteiro et al. (2008)

• Motorola WEAVR (2007)

De Fraine et al. (2008)

• Management of aspect interactions using statically-verified control-flow relations
  – Control-flow interactions
    • Typical aspects in a multi-tier architecture
      – Presentation, logic and data tier
      – Caching, authentication and authorization aspect
    • No shared join points
De Fraine et al. (2008)

• Management of aspect interactions using statically-verified control-flow relations
  – Produce abstraction of the possible control-flow paths in the resulting application by statically analyzing application code and woven aspects
  – Document aspects with control-flow policies that express valid and unvalid control-flow relations that the aspects depend on
  – Use existing static analysis techniques to automatically verify the policies against the produced abstract paths

Monteiro et al. (2008)

• Towards an analysis of layering violations in AO software architectures
  – Layering violations
    • Skip call
    • Back call
    • Cyclic dependency
  – Identify what are the layers and to which layer each module belongs
  – Violation measurement framework in order to detect all dependencies between modules
Motorola WEAVR

- AOM weaver developed at Motorola able to weave UML statecharts
  - Focus: specify aspect precedence explicitly at the modeling level in order to reduce occurrences of aspect interactions
    - Advice – advice
      - <<follows>> relationship
    - Aspect – aspect
      - <<follows>>, <<hidden_by>> and <<dependent_on>> relationship


Architecture and design

- De Fraine et al. (2008)
- Monteiro et al. (2008)
- Motorola WEAVR (2007)
Middleware

• Lupu et al. (1999)
• Wohlstadter et al. (2004)
• Zambrano et al. (2006)
• Greenwood et al. (2007)
• Sanen et al. (2007)

Lupu et al. (1999)

• Conflicts in policy-based distributed systems management
  – Focus
    • Offline conflict detection and resolution
    • Authorization and obligation policies
      – what activities a subject can perform on a set of target objects
      – what activities a manager or agent must or must not perform on a set of target objects
    • Precise notation, logic-based

Lupu et al., Conflicts in policy-based distributed systems management, 1999.
Lupu et al. (1999)

- Two kinds of conflicts
  - Modality
    - When both positive and negative policies apply to the same objects
    - Detection: purely syntactic analysis
    - Resolution: establishing a precedence
  - Application specific
    - Sort of meta-policies: constraints about permitted policies (e.g. to take limited resources into account)
    - Only experimental so far

Lupu et al., Conflicts in policy-based distributed systems management, 1999.

Wohlstadt et al. (2004)

- Middleware-based approach to managing dynamically changing QoS requirements of components
- Use of policies to advertise non-functional capabilities
- Provides middleware support to match, interpret and mediate QoS requirements of clients and servers at deployment and/or run time as a main contribution

Wohlstadt et al., GlueQOS: Middleware to sweeten QoS policy interactions, 2004.
Wohlstadter et al. (2004)

- Assumption of a fixed ontology of features with all interactions explicitly identified ahead of time
- Task of handling feature interactions separated out in policy mediator

Wohlstadter et al., GlueQOS: Middleware to sweeten QoS policy interactions, 2004.

Zambrano et al. (2006)

- Solving aspectual semantic conflicts in resource-aware systems
  - Focus: aspects that reify resource awareness concerns in an AO middleware context for mobile applications
  - Explicit focus on conflicts that exist even if aspects are not working on the same join points
  - Examples
    - Memory saver vs. battery optimiser (memory issue)
    - Network optimiser vs. security vs. processing time (bandwidth and processor issue)

Zambrano et al. (2006)

- Conflict resolution
  - Semantic labels (denote role played by aspect w.r.t. resource)
    - Descriptors added to aspects that expose aspects’ metadata indicating the kind of resources utilised and the way they are affected, cfr. Java annotations
      - Memory/bandwidth/processor
      - Consume/release
  - Coordinator aspect to control execution
    - Consuming semantic labels
    - Set of strategies as operations (rule-based) on aspects


Greenwood et al. (2007)

- Interactions in AO middleware
  - Focus: specify aspect interaction contracts that are enforced at runtime
  - Interaction model based on shared elements
    - Common join points, component instance or base
    - Meta-data
  - Validated in custAOMWare
    - Prototype AO middleware platform

Greenwood et al. (2007)

• Aspect interaction contracts
  – Basic
    • Requires [aspect, meta-data, …]
    • Provides [aspect, meta-data, …]
  – Advanced
    • Conflict [aspect, meta-data, …]
      – E.g. Conflict(thread[AC_2]) in AC_1’s contract
    • Precedence [aspect, meta-data, …]
      – E.g. Precedence(type[AC_1, AC_2])
    • Resolution [aspect, meta-data, …]
      – E.g. Resolve(instance[AC_2 AND AC_3]) in AC_1’s contract

Sanen et al. (2007)

• Composition of common middleware services by an application developer requires understanding of many implementation-specific details
• Even higher complexity because various common middleware services interact with each other
• Adding new services to an existing middleware may break it’s behavior or the behavior of the application on top of it
Sanen et al. (2007)

- Knowledge driven management of interactions to provide tool support prescribing to the developer what are the important decisions to take in a composition
  - Conceptual model for explicitly representing knowledge about concern interactions
  - Use reasoning techniques to provide practical support to the software developer for managing interactions
  - Prototype implementation


Middleware

- Lupu et al. (1999)
- GlueQOS
- Zambrano et al. (2006)
- Greenwood et al. (2007)
- Sanen et al. (2007)
Language support

• Precedence
• Interface-based approaches
• Pawlak et al. (2005)
• Durr et al. (2006)
• Katz et al. (2007)

Precedence

• When multiple aspects apply to the same join point, different composition orders may reveal various inconsistency problems
• Interactions can be identified through static analysis
• Resolve interactions by ordering aspects
  – Light-weight support by declaring precedence
    • E.g. AspectJ
  – More complex dependencies and ordering relationships
    • E.g. Kienzle et al. (2003)
  – Advanced approaches that require extra behavior specifications from the user for each advice or each aspect
    • E.g. Durr et al. (2005), Pawlak et al. (2005)
    • E.g. Sihman et al. (2003), Lagaisse et al. (2004)
Interface-based approaches

- Open modules (Aldrich et al.)
  - Calls to interface functions from outside the module can be advised by clients
  - Clients can advise exported pointcuts
- Crosscutting interfaces (Sullivan et al.)
  - Expose behaviours through interfaces
  - XPI consists of a name, a scope over which the XPI abstracts join points, one or more sets of abstract join points, and a partial implementation.
- Aspect-aware interfaces (Kiczales et al.)
  - Aspects extend the interfaces of modules they advise
  - Dependencies of an aspect to a join point are computed and added as annotation to the interface of advised code

Pawlak et al. (2005)

- CompAr: language that allows programmers to abstractly define
  - execution domain
  - advice codes
  - often implicit execution constraints.
- High level of abstraction
  - Very generic advice definitions 😊
  - On a per join point basis 😕
- Automatic detection and solving of aspect-composition issues (interactions between aspects) of around advices

Pawlak et al., CompAr: Ensuring Safe Around Advice Composition, 2005.
Durr et al. (2006)

- Language-independent technique to detect semantic conflicts among aspects that are superimposed on the same join point
- Semantic conflict: emerging behaviour that conflicts with the originally intended behaviour of the involved components
- Example of logging and encryption applied to same join point

Durr et al., Reasoning about semantic conflicts, 2006.

Durr et al. (2006)

- Argue that we need a means to express aspect behaviour and conflict detection rules
  - Abstraction mechanism to represent essential behaviour of an aspect and sufficient to detect semantic conflicts
- 3-step resource-operation model
  - Key idea: some resource must be shared among aspects for the aspects to conflict
    - Automatic listing of advices per shared join point
    - Resource essentially an abstract data type with set of operations
    - These tuples are matched against conflict detection rules

Durr et al., Reasoning about semantic conflicts, 2006.
Katz et al. (2007)

- Describe interference among aspects at the semantic level, irregardless of any overlap among join points or variables.

- Semantic interference: when one aspect can make another aspect operate incorrectly even though each of both works correctly in isolation.

- Examples
  - Security: encrypting sent pswd / retrieving forgotten pswd
  - ATM: return card after failure / withhold after 3 wrong pswd
  - Messages: sent numbers limited size / encrypting them

Katz et al. (2007)

- Aspect specification
  - Aspect = advice + introductions + pointcuts
  - Aspect specification
    - Assumptions: preconditions about system without aspect
    - Guarantees: result assertions about system with aspect
  - Specifications are written in temporal logic (TL)
  - Assumed that each aspect has specification and is correct w.r.t. that specification

- Interference and aspect categories
  - Spectative, regulative, weakly invasive and strongly invasive aspects
  - Optimizations possible if two spectative or regulative aspects interfere
Language support

• Precedence
• Interface-based approaches
• Pawlak et al. (2005)
• Durr et al. (2006)
• Katz et al. (2007)

Full lifecycle support

• Phd Beuche (2003)
  – Composition and construction of embedded software families
  – CONSUL approach for family-based software development
    • Tool support for every related activity in all phases of the development lifecycle
    • Using many concepts already known like feature models and CBSD
  – http://www.pure-systems.com
Illustrated

CONSUL enables a user to select any feature she or he likes and can immediately execute checks whether there are open decisions or unfulfilled restrictions.

Process

• Determining valid combination of features
  – Pure::variants checks whether the user selection is valid and resolves dependency conflicts automatically
    → Feature combination

• Selecting a suitable solution
  – Using a component family model (for mapping user demands to component implementations) and artificial intelligence techniques
    → Component description

• Creating the solution
  – Customized solution created by transformation process controlled by the component description
Outline

- Problem statement
- A historic perspective
- State of the art
- Towards the future
Outline

• Problem statement
• A historic perspective
• State of the art
• Towards the future

Some conclusions

• Interactions are a complex, but interesting research topic
• Interactions are relevant across the entire software development lifecycle
• Different measures can be taken during different lifecycle phases
Challenges ahead

• Cope with “semantic” interactions
• Towards the right abstraction
• Explaining interactions
• Collecting interactions

Cope with “semantic” interactions

• Lots of work exists on syntactic interactions
  – E.g. Aspects sharing join points
  – E.g. Aspects manipulating the same variables
• What about “semantic” interactions
  – Purely behaviour-based, no shared join points or variables
    • Check behavioural abstractions for inconsistencies?
    • Document interactions themselves in a generic way so that this information can be shared and (re)used?
    • Specify contracts that take care of everything?
    • ...
Towards the right abstraction

• What is the appropriate level of detail?
  – Dependent on the lifecycle phase you are in?
• Express what is allowed or what is not?
  – Is it possible to list everything that is not allowed?
  – Is it realistic to come up with a complete and 
    correct specification of what is allowed?
  – Do we need a combination of both?
• Inherently incomplete knowledge
  – New interactions always can and will arise

Explaining interactions

• Full lifecycle awareness for interactions
  – E.g. guiding developers through middleware composition
  – E.g. notifying an application user of an occurring interaction and asking for a decision
• How to explain an interaction to any relevant parties?
Collecting interactions

- Examples are used over and over again
  - E.g. logging and encryption
- Most approaches assume a fixed list of interactions to start from
  - How realistic is this? Non-trivial process!
  - How can we systematically collect interactions?
    “Analyzing feature interactions for all possible feature combinations and having the components ready for them is probably too difficult.” [Kang et al. (2002)]

Towards the future

- Questions to ask yourself
  - Do I believe in complete and correct specifications of every module in a system?
  - How do I feel about formal methods?
  - Do I believe that all kinds of interactions should be prevented or would I prefer detecting and resolving them at runtime?

Come talk to us!
Questions?

• Thank you for your attention!