

Does AOSD improve the ability to reason about the correctness of a system?

Panel Chair: Awais Rashid (Lancaster University, UK)

Panelists: Adrian Colyer (IBM Hursley Labs, UK)

Lodewijk Bergmans (University of Twente, The Netherlands)

Klaus Ostermann (Technical University of Darmstadt, Germany)

Elke Pulvermueller (University of Karlsruhe, Germany)

Adrian Colyer

For small, simple programs, it doesn't really matter how you construct your application - it will be relatively easy to understand what's going on. So let's say that we are interested in larger software systems, probably developed by a team and maybe through several releases.

Often for these systems it is simply not possible for any individual to understand every detail of their construction. We understand and reason about them the same way we do all complex systems - by applying a divide and conquer strategy to understand the parts of the system in isolation, and then putting the pieces together to form a whole. AOSD supports this line of reasoning fully. First the main path logic can be understood in isolation, free from consideration of other concerns (that is to say, tangling introduces cognitive overhead and distracts from the main flow of concentration). Then, each supplemental concern can be studied and understood in isolation. Here AOSD is of tremendous benefit because an aspect can state clearly not only what behaviour it implements, but also where that behaviour should apply. The switch from a scattered implementation to an explicit statement of policy makes the structure significantly easier to reason about.

Enforcement of the aspect policy through the use of declare error or warning constructs gives confidence that the policy is indeed faithfully applied throughout the system.

When it comes to putting the parts together to form a whole, the aspect declarations provide a succinct and clear specification of how the pieces should integrate. So scattering and tangling, which are greatly reduced by the application of AOSD, make it harder to reason about a system, and explicit statement of cross-cutting policy (as captured by an aspect) makes it easier to reason about a system.

The final piece of the puzzle is tool support, which enables a developer working with a piece of system code to be fully aware that once the main-line logic in front of them has been understood there are additional concerns to be investigated. A tool such as AJDT can provide direct linkage to the definition of those concerns from the editor buffer.

Lodewijk Bergmans

My focus is on the problems that are caused by interference between aspects. This is also referred to as the aspect interaction problem. These problems may occur when multiple aspects affect the same base modules or join points: due to physical and temporal distribution of module development, this may be unnoticed by the developers, but it is not unlikely that the combination will result in conflicts. Hence AOP introduces new ways that the correctness of a system can break down.

The core motivation for this work: the introduction of new aspects may cause certain modules to cease functioning as expected, due to conflicts, or interference, between the superimposed aspects. This is particularly inconvenient, since the occurrence of the problems may be unpredictable to the responsible developer(s): these aspects can be introduced independently, also at different times, and the programmer of each aspect may be unaware of the existence or future introduction of the other aspect(s).

One of the general difficulties that AOP brings, which make it harder to reason about the correctness of systems, is that no definite correctness observations about a particular piece of software can be made without full global knowledge (i.e. the assurance that no aspect superimposes some behavior/advice in me that breaks down my correctness); this is due to the ability to specify crosscutting in combination with dependency inversion.

Klaus Ostermann

There can be no doubt that AOSD makes it easier to reason about individual concerns of a system because their implementation can be localized better and is less blended with the implementation of other concerns. However, in contrast to the improved locality and modularity of the individual concerns, AOSD languages tend to destroy the locality of the run-time control flow - we can no longer deduce the flow of control in a module and its interaction with other modules by looking at the module only. We have seen the same phenomenon (to a much lesser degree, though)

with the introduction of dynamic binding in object-oriented languages.

Many tools such as AJDT use static analysis in order to infer the interactions between the modules statically. Although these tools are undoubtedly useful, static analysis is only possible for very static, syntax-directed join point languages. For the next generation of AO languages, I envision much more dynamic, powerful join point languages - hence I believe that static analysis will not be an option in the long run.

However, this does not mean that we are doomed! Let us consider briefly, *_why_* we want to analyse the extent of join points and hence the flow of control: I think the reason is that we want to check whether our join point specification matches our *_intention_* of the join point. The problem is that current join point languages are too low-level - they don't allow us to state our intention directly, but instead we have to encode it more or less as expressions over the syntax tree. Once we have more precise and more intentional join point languages, there will be no need for that kind of analysis because we *_know_* that the behavior is as desired.

Elke Pulvermueller

Main thesis: Aspect-Oriented Software Development and system correctness are orthogonal to each other.

Consequence: AOSD does not per-se improve correctness.

Examination of AOSD: Having a closer look at AOSD we determine a wide range of its realisation. These realisations range, for instance, from static approaches (e.g. Aspect/J) where the compiler gets the full knowledge to produce the end system to fully dynamic ones (e.g. eAOP) where the actual control-flow determines the aspect to be called.

Despite all such differences, however, several properties are typical to all realisations:

- the join point model is a meta-description (control-flow detection patterns plus semantics)
- an aspect is distributed into the control-flow in the running system and as such realises invasive composition
- AOSD systems are usually composed of various, very fine-grained aspects in practice
- the AOSD system structure supports configuration flexibility; aspects may be exchanged easily

AOSD offers a next step to OO improving modularisation on a more fine-grained level. Aspects offer constructs allowing a one-to-one mapping to system/requirement features.

Relationship of these properties to correctness considerations:

- The additional meta-descriptions are an additional source of errors (e.g. problem of weaving order; e.g. filter order in CF).
- For the developer it becomes difficult to keep the overview of the run-time aspect which is an additional source of errors.
- Invasive changes are contradictory to pure black-box approaches allowing as such "uncontrolled" changes similar to the fragile base class inheritance problem.
- The "distributed" aspects are a challenge for the analysis of the program. On the other hand a sound description of join points limits uncontrolled changes to specified program elements (e.g. only before / after a method) allowing a gray-box approach instead of a pure white-box approach.
- The high number of fine-grained aspects improve the realisation of systems (supporting the divide-and-conquer approach). Small units are easier to produce than large monolithic systems. This affects the probability of errors the developer inserts into these units. On the other hand the complexity increases due to the large number of aspects which have to be composed properly.
- Configuration is a general problem with respect to correctness. Means are necessary supporting to prove correct configurations. Manual approaches soon reach the limit of manageability. Similar as to the problem of various fine-grained aspects computer-support is needed in dealing with a large number of units and their composability.

Summary and conclusion: AOSD improves the flexibility to express a program by allowing invasive changes at more or less limited program locations. It also improves the potential for modularity and flexible system configuration. These two issues (invasive changes and modularity), on the other hand, bear a challenge for correctness-ensuring techniques.

We have additional challenges in reasoning about the correctness of AOSD systems. Therefore, we need appropriate models for aspects and aspect systems abstracting from the concrete system and allowing to verify specific properties. Basic techniques for this task are already available (ECOOP WS#11: Correctness of Model-based Software Composition) and ready to be used in AOSD. The primary goal is not to reinvent correctness ensuring techniques but first exploit existing ones.