Aspect-oriented programming for distributed systems: its use, its effect on language design, and its limits

Peter Van Roy
Université catholique de Louvain

May 3, 2004

Belgian Symposium and Contact Day
on Aspect-Oriented Programming and Software Evolution

Overview

- Distributed programming
- AOP in depth
- Distribution structure
- Fault tolerance
- Security
- Consequences for the language
- What have we achieved so far?
- The limits of AOP

Building real distributed applications

- Building distributed applications involves many different concerns: application functionality, distribution structure, tolerance for partial failure, security, and so forth
- Instead of trying to build a generic tool that can handle all these concerns, we will **treat each concern in depth** (solve it completely) before going to the next.
- We will see that language design is an important part of the solution. We will change the language semantics when necessary. We use a language that is expressive, simple, and has a complete formal definition.
- Note that we are not doing AOP in the AspectJ/MOP tradition of "weaving" source transformations. It may be possible to transfer some of our work to such a context (although we remain skeptical). We leave this question up to you!

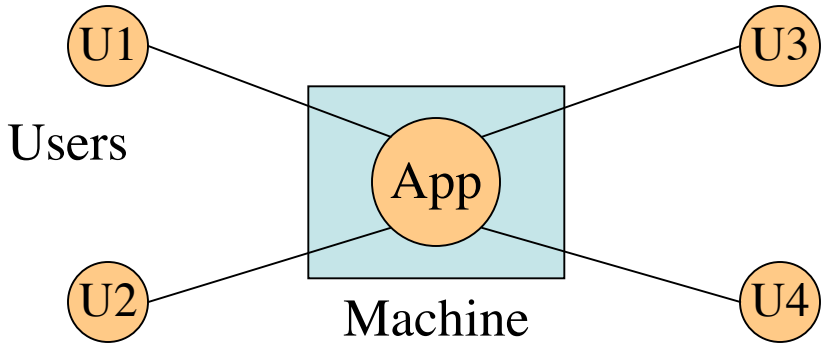
AOP in depth

- We consider that AOP should ideally be programming with **conjunctions of partial specifications**
  - See "Understanding Aspects", Mitchell Wand, August 2003
- The complete specification consists of several parts that are logically "anded", such as:
  - Application functionality and resulting architecture
  - Distribution structure and performance
  - Fault model (partial failure) and fault tolerance
  - Threat model and security
- Crucial property: **each new part should not invalidate the others**!
  - We shall see that this can be achieved. However, this does not mean that there is no interaction between the parts! For best results in each part, the application architecture must be designed accordingly.
- Each part has its own (small) domain-specific language (DSL). The parts are independent as far as possible, but there will be some interaction between them (e.g., keeping centralized performance in a distributed application requires some changes to application functionality and architecture). This interaction cannot be avoided, but it can be minimized and compartmentalized.

Example of programming with partial specifications

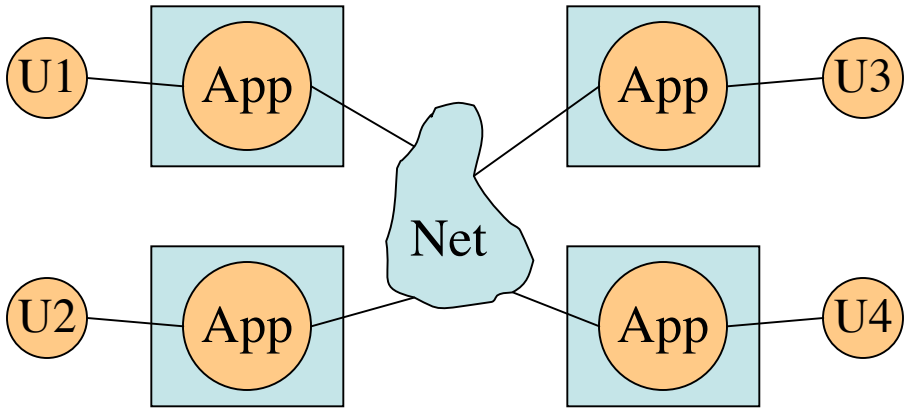
- Application architecture
  - Client/server chat program where each client and each server is modeled by one object
- Distribution structure
  - Clients and server are stationary objects on separate machines; each object invocation results in one round trip message
- Fault tolerance
  - Fault model: machines can crash with fail stop
  - Server object stores all state and is replicated using primary/secondary algorithm; client objects are not replicated
- Security
  - Threat model: server is trusted; client must be authenticated to be trusted; network is untrusted
  - Client/server communication is encrypted; all clients can see all messages; all state is stored securely at the server

Another example: a collaborative drawing tool



- To fix the ideas we give another example, a simple collaborative application that is a drawing tool with several simultaneous users
- The ideal case for this application is that functionality is the only aspect: the application is written as if it were running on one machine that is reliable and all users and the machine are trusted.
- We will extend this application in steps to the distributed case. This causes the other aspects to appear naturally.

Adding distribution structure



- We partition the application over multiple networked machines. We assume network transparency: the language semantics is unchanged independent of how the partitioning is done (note that we are not handling partial failure yet).
  - Network transparency follows from the fact that adding a partial specification (of distribution structure) does not invalidate the original specification (of functionality)
- We then add a partial specification giving the distribution behavior of the resulting partition
  - This is how performance is regained
  - For example: some of the application objects will be shared, that is, have remote references in the partitioned application. We give these objects a distribution semantics, for example, as stationary object (standard server style), cached object, or invalidation object.

Consequences of network transparency for language properties

- For network transparency to be practical, the language must allow for efficient distribution. This has an effect on how the language expresses state and concurrency.
- **State (a.k.a. destructive assignment)**
  - State is expensive in a distributed setting, since updates must be globally consistent
  - The language should therefore make it easy to program with weaker forms of state (stateless/immutable, single assignment, or localized state)
- **Concurrency**
  - Distributed systems are naturally concurrent; imposing a global sequential semantics is expensive
  - The language should make it easy to program with various forms of concurrency
- Conclusion: in a distributed setting, ***the natural behavior is stateless and concurrent***. This should also be natural in a language intended for distributed programming!
  - One the other hand, state and sequentiality do have advantages: we do not want to jettison them completely. They are used primarily in a centralized setting.
  - A language for distributed programming should be agnostic with respect to state/stateless and sequential/concurrent programming

Language design is essential!

- It follows that the problem of doing application development in a distributed setting cannot be solved without doing some **serious language design**!
  - The usual languages don't cut it. Usual OO languages (Java, Smalltalk) put too much emphasis on state and make concurrency difficult. Usual functional languages (Ocaml, Haskell) are better, since they realize the importance of stateless behavior, but they don't support stateless concurrency.
  - This is one of the main goals of the Mozart Consortium since 1995
- Two main results:
  - Language design: **Oz**, a language that treats state and concurrency in a careful way with an efficient implementation and a full formal semantics
  - System building: **Mozart Programming System**, a platform that implements network-transparent distribution with distributed algorithms
- I won't say so much about Oz and Mozart specifically in this talk
  - For more information, I recommend the official Web site (`http://www.mozart-oz.org`)

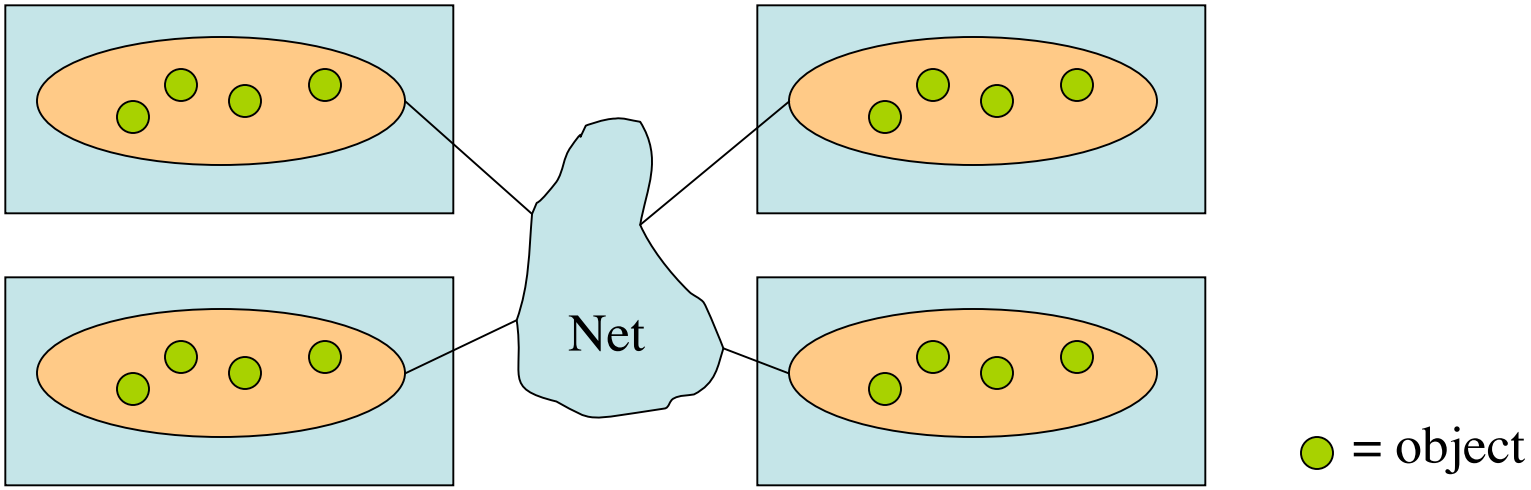
Our language insights…

- The best place to learn about our language insights is the book "Concepts, Techniques, and Models of Computer Programming", by Peter Van Roy and Seif Haridi, just released by MIT Press in March 2004
- It "deconstructs" Oz by showing why and how each new concept is needed, for example:
  - **Declarative concurrency (chapter 4)**: a form of concurrency without race conditions; programming that is stateless and concurrent. We extend this to the two well-known forms of concurrency, message-passing and shared-state.
  - **Data abstraction (chapter 6)**: the different ways to do data abstraction (object style versus ADT style). The role of state in data abstraction has to do with modularity, which is the ability to change one part of a program without changing the rest.
  - It gives an extremely simple formal semantics for all the concepts (chapter 13)
- It also shows how transparent distribution works in Oz (chapter 11)
- We are using it as a textbook for teaching programming
- See `http://www.info.ucl.ac.be/people/PVR/book.html`

Giving objects a distribution semantics

- The distributed semantics of shared objects is a partial specification, an "aspect"
- Simplest case
  - **Stationary object**: synchronous, similar to Java RMI but fully transparent, i.e., automatic conversion local↔distributed
- Next step: tune distribution behavior *without changing application architecture*
  - Use different distributed algorithms depending on usage patterns, but application architecture is unchanged
  - **Cached (« mobile ») object**: synchronous, moved to requesting site before each operation → for shared objects in collaborative applications
  - **Invalidation-based object**: synchronous, requires invalidation phase → for shared objects that are mostly read
- Final step: tune distribution behavior *with possible changes to application architecture*
  - In most cases changes are unavoidable, e.g., to overcome large network latencies or to do replication-based fault tolerance (more than just fault detection)
  - **Asynchronous stationary object**: send messages to it without waiting for reply; synchronize on reply or remote exception
  - **Transactional object**: set of objects in a "transactional store" supports local changes without waiting for network (optimistic or pessimistic strategies)

Transactional objects with the GlobalStore abstraction



- The GlobalStore is transparent; it looks like a set of objects with a transactional interface
  - The application affects the GlobalStore through transactions
  - The GlobalStore affects the application through notifications
- The objects are replicated on each machine, which also provides fault tolerance and latency tolerance!
  - Current implementation handles only machine failure; extension for network partitioning is in progress

Adding fault tolerance

- The GlobalStore is an example of the next aspect: fault tolerance
- Fault tolerance is important because of a new behavior that appears when distributing an application: partial failure (part of the system fails; we would like the rest to keep working)
- Fault tolerance is solved by a similar approach as distribution structure
  - Fault tolerance can be provided (up to a point) by adding a partial specification, without changing the application architecture. For example, we could provide replication for all the application's objects.
  - This is expensive, however. To improve efficiency, the application architecture has to be changed.
  - The GlobalStore is an example of this: once the decision to use a transactional object store is made, fault tolerance comes "for free", since we can add replication transparently
- How do we build fault tolerance libraries within the language?
  - The language should make it easy to build these libraries. The language semantics needs to be extended to give well-defined behavior in the case of faults and to provide reflective fault detection.

Fault detection within the language

- Fault tolerance abstractions can be built with reflective fault detection
- **Reflective** fault detection
  - Reflected into the language, at level of single language entities
  - Pragmatic fault model: **permanent process failure** and **temporary network failure** (these are by far the most common on the Internet)
- Both synchronous and asynchronous detection
  - Synchronous: raise exception when attempting a language operation
  - Asynchronous: when a fault is detected, then start a user-defined operation in a new thread. Attempted language operations block indefinitely.
  - Our experience: **asynchronous is better** for building abstractions. The GlobalStore is built with asynchronous fault detection. We have not found any use for synchronous fault detection.

Adding security

- The next aspect is security
- The ideal case is where everybody trusts everybody; users are trusted and the execution environment is trusted
- The application deviates from this in two ways:
  - It is **multi-user**: users have to be protected from each other (malicious or careless users)
  - It is **multi-domain**: when distributed, the application may execute over a distributed system whose parts are owned by different organizations and which may be used by third parties (either malicious or careless)
- The first step to handle this is to define a threat model
  - The next step is to counter the threats
- Again, the language design is important!
  - The language should make it easy to build secure applications. What does this mean?

Security within the language

- A first principle is that the language should make it possible to write secure applications within the language, i.e., for adversaries that stay within the language
  - This means that the language should have a formal semantics and be based on abstract entities, that is, entities that are completely defined by a set of abstract operations
  - Many high level languages satisfy this property up to a point (e.g., Java and Smalltalk). C and C++ do not (since they allow access to machine representations).
- Programming in such a language is in fact programming with capabilities, where a **capability** is a reference that combines two properties: it designates an entity and it confers authority to perform some operation on that entity (Dennis & Van Horn 66)
  - In principle, any abstract entity can be a capability (integers, functions, classes, …)
  - In particular, object references are capabilities
- With capabilities, we can structure the application according to the **Principle of Least Privilege** (the "need to know" principle): each part of the application only has authority to do what it needs to do, and no more
  - A happy discovery is that standard object-oriented programming techniques already go most of the way! In fact, supporting Least Privilege fully means that there can be absolutely **no cheating** regarding these principles.
  - The designers of the **secure language E** find that this is practical (Mark Miller et al)

Security outside of the language

- Having a secure language is only the first step
  - Two additional steps are needed
- A **secure implementation**, to ensure that language security cannot be broken by adversaries that can access the implementation (e.g., using cryptographic techniques when necessary)
- A **security architecture**, to ensure that the application handles security as it should (e.g., authorizing users and keeping track of what they have rights for)
  - One important abstraction, discovered during E research, is the PowerBox. This generalizes the sandbox to allow dynamic negotiation of additional capabilities needed during execution.

What have we achieved so far?

- We have given an approach to distributed programming with conjunctions of partial specifications, where each partial specification is an aspect
  - Application functionality
  - Distribution structure
  - Fault tolerance
  - Security
- A crucial insight is that designing a system to handle these aspects also has **consequences for the language design**
  - Regarding state and concurrency, well-defined behavior for partial failure (including reflective fault detection), and formal semantics with abstract entities (allowing programming with capabilities)
- Another crucial insight is that **the aspects are interdependent through the application architecture**!
  - Some (small) headway can be made by adding aspects independently without changing the application architecture
  - But realistic application design means that adding an aspect fundamentally requires modifying the application architecture. This means that other aspects are modified indirectly as well.
  - However, the aspects can still be understood independently of each other, since each partial specification only specifies its particular aspect

The limits of AOP

- There are other important aspects we haven't talked about yet:
  - Evolution (maintenance, versioning, hot code replacement)
  - Openness (how do independently written applications find each other and exchange information)
- Together with application functionality, distribution structure, fault tolerance, and security, this makes for a lot of aspects!
  - For each new aspect the architecture has to be changed
- Complexity increases, even with the **best possible AOP**!
- Solution: **seal off** some of the aspects (e.g., like virtual memory)
  - Example: GlobalStore seals off distribution structure and fault tolerance
  - Example: a generic "hierarchical service architecture" (Bruno Carton)
    - Module graph per machine with synchronous calls & hot module update (using atomic state transfer protocol)
    - Decentralized overlay network to link machines, with asynchronous calls
    - The idea is to generalize the GlobalStore to be decentralized and no longer just a "flat" object store

Work on the Oz language and the Mozart system

- We are doing long-term research to implement these ideas for Oz and Mozart
- Mozart 1.3.0 provides almost perfect network transparency and good specification of distribution behavior
  - The DSS (Distribution Subsystem), planned for the next release in early 2005, will make both perfect (Erik Klintskog)
- Mozart 1.3.0 provides fault detection and well-defined behavior in the case of partial failure (machine failure and network partitioning)
  - Work on abstractions to exploit this is still in progress (the GlobalStore is working but still a prototype and does not yet handle network partitioning; the P2PS library, a decentralized self-organizing overlay network, has been released)
- Oz provides language security (complete formal semantics and capabilities)
  - Implementation security and security architecture are still work in progress
  - We are collaborating with the E designers to realize this
- We are working on good abstractions to seal off some aspects
  - Still very much ongoing, only preliminary results so far

Conclusions

- AOP "in-depth" means programming with **conjunctive partial specifications**
  - Each specification is written in a DSL
  - No magic bullet; each specific aspect requires thorough research
- Aspects are formally independent, but are **interdependent through the application architecture**
  - The application architecture has to be tailored to allow for each
- Even the theoretically best possible AOP is **not good enough**
  - Need to "seal off" aspects to program at higher levels
- Our research vehicle at UCL is the Mozart system. We focus first on functionality, distribution structure, fault tolerance, and security
  - An important part of this research is **language design**
  - We have written a textbook that distills our language insights; we are using it for teaching programming
- For more information, please come to the **MOZ 2004** conference!
  - Oct. 7-8, 2004 in Charleroi, see `http://www.cetic.be/moz2004/`