Statically typed temporal scopes

Marc Girod

Nokia Telecommunications
Kilo RD4
P.O. Box 12
02611 ESPOO – Finland

Email: marc.girod@ntc.nokia.com
Tel: +358–0–511 27703
Fax: +358–0–511 27432

Statically typed temporal scopes

The benefits expected from distribution are commonly listed as:

- Accessibility and availability of resources.
  The unity of the system must be preserved. Let's insist on the duality: availability *here* and *now.*
- Scalability
  Every system imposes some limits (resources, performance) to scalability. Distribution is a way to overcome these.
- Fault–tolerance
  Distribution allows for redundancies, which may be used to support resilience to local failures, or cross–checking of automated decisions.
- Flexibility
  Distribution offers potentialities for local maintenance, backup or updates, even at run–time. This is valuable e.g. for systems that cannot be disconnected from the resources they manage, for fear of loosing track of them.
- Efficiency
  Concurrency may, in many cases, improve performance.

These targets may be desirable for themselves, directly, but also indirectly, in order to reach some other goals, such as:

- Extensibility, which implies in practice scalability
- Robustness, which can be served by fault–tolerance
- Reusability, requiring accessibility
- Modularity, etc.

The list given above is thus not unique. Its components are not orthogonal either: a correlation between the size of a system and loss of efficiency will set an upper limit to scalability, for instance.

Nevertheless, this list provides, better than a definition would, with a specification for what is at stake as one speaks of distribution.

In order to support scalability, we must provide *late distribution*: i.e. allow to reconfigure the distribution without invalidating client software.

A first step toward this goal is to provide *location transparency*, but there are more fundamental reasons why traditional sequential software is not distributable.

Distributing sequential code impacts on its semantics.

- Reachability becomes an issue: the service will depend on the network. Remote objects may be OK, but non–reachable.

- Non–determination grows: the state of remote objects may change without notice, so as to alter or invalidate their ability to provide the service expected by their clients. One cannot accept the raise in non–determination since it impairs availability (or scalability at constant availability, or efficiency...). On the other hand, one cannot expect clients to monitor the environment.
- Dead–locks may occur, resulting in non–availability, either temporary, because of the interference of other processing, or definitive, when resulting from reproducible conflicts, intrinsic to the distributed configuration of the chained implementation of a service.

All these additional issues alter the semantics of the service contract. Furthermore, these alterations tend to spread along use chains, from servers to clients, and end up typically in affecting higher–level services most!

A typical strategy to minimize this paradigm shift is to settle for synchronous RPC (e.g. in CORBA — asynchronous *oneway* calls are also provided; we shall ignore this option —). Synchrony is incompatible with the requirement for availability, applicable to object servers. In the event driven architecture of CORBA servers, asynchrony is commonly restored by using threads.

The shortcomings of this situation seem to appeal for transactions. With them, we are pretty close, but are we quite on the track?

First, let's note that basic transactions are not enough. To be up to the challenge, transactions must in addition:

- be composable: either nested or implicit (no non–nested transactions in reusable client code!)
- provide checkpoints for partial rollbacks (otherwise, the costs accumulate by composition)

Yet, transaction semantics are both...

too weak:

- there is no way to precisely compare two transactions with respect to their use of a resource, which typically results in wrong diagnoses of conflict;

and too strong: many behaviours are just not suitable for being modeled as transactions, because

- they cannot be rolled–back (e.g. action onto a physical device, output)
- they don't ever "complete" (e.g. maintaining a representation)

Temporal scopes

I argue that the concept we are missing is this of *temporal scope*.

Scope is the syntactic unit of locality.

It offers boundaries for modules. If these boundaries are explicitly declared, they may be checked. This explicit declaration is what typing is all about. The existence

of scopes allows for a synthesis between polymorphism and static typing. This gives us some control on dependencies, and thus makes the decoupling between client and server code possible.

In procedural languages, as C, the only named scopes were functions. C++ extended this concept to classes.

A function call encapsulates two messages (initiation and completion) and a scope. The scope is usually hidden. It encompasses the actual transition from the *before* to the *after* state — the actual state transition is hidden, so that it may be defined in different ways, without invalidating any client code.

If a (member) function is specified in the client context as to trigger other virtual functions, it is binding for the implementations which will define these virtual members. It works therefore as a behaviour specification, in a practical partial (abstract) way, without the burden of formal specifications.

With distribution however, function scopes break down: their traditional acception is badly modified by the introduction of *time*.

Sequential programming never deal with *concurrency*, and largely ignores *duration*. Furthermore, *causality* is hidden under mere sequentiality, clearly an over–specification. I would say that sequential programming paradigms are largely timeless.

This is what is not admissible with distributed programming.

What we need is an explicit support for temporal scopes. Instances must be identifiable so as to be kept up–to–date (reactivity). These scopes must be modeled as classes. I call them *operations*. Their implementation involves a large amount of genericity, i.e. templates.

The operation interfaces are dedicated to their specific ”peers”. Operations have interfaces to:

- the request broker: low–level interface, only in the concrete implementations
- the operation’s initiator: at various depth of specialization
- other operations: idem

In what does this help?

Asynchrony

- Operations have an identity (instances), and thus may be notified asynchronously, and kept up–to–date. This way, their advertised behaviour may be kept deterministic.
- Because of determinism, there is no penalty in performance or availability at the time of use, i.e. for the clients. The impact of distribution is smoothened.
- Early detection of conflicts, due to the registration of operations at the resources (advantages of ”optimistic locking” without its inconveniences).

- Asynchronous notifications allow for safely making local copies for efficiency, or fault–tolerance.

Typing and composition

- Operations offer an arbitrarily fine granularity, and semantics to minimize the actual conflicts. This enhances availability and efficiency.
- If the operations are specialized for the actual resources, conflict detection can be performed locally, but the arbitration may access information from the ultimate initiators, and thus decrease the impacts of eventual (often partial) roll–backs

Conclusion

One major achievement of object–orientation is the move from the global scope to programming *in–context*. This is worth preserving in moving forward to distributed programming.

Related work

Atria ClearCase™. *Comment*: support for parallel development in a configuration management system. The product maintains implicitly consistent client *views*. These views are largely based on the notion of *branch*.

Systematic Concurrent Object–Oriented Programming, by Bertrand Meyer, in Communications of the ACM, September 93. *Comment*: Meyer suggests an extension to Eiffel, but this breaks the seamlessness of distribution.

Higher–Order Distributed Objects, by Henry Cejtin, Suresh, Jagannathan, and Richard Kelsey, in ACM Transactions on Programming Languages and Systems, September 1995: Vol 17, Nr 5). *Comment*: untyped support for scopes.

CHSM, by Paul J. Lucas: a Concurrent, Hierarchical, State Machine language system that extends C++ for implementing reactive systems. *Comment*: based on Harel statecharts. These are interesting because they support polymorphism on states, and thus are extensible.

Regular Types for Active objects, by Oscar Nierstrasz, OOPSLA ’93. *Comment*: more traditional work on FSM.

Interfaces, Protocols, and the Semi–Automatic Construction of Software Adaptors, by Daniel M. Yellin and Robert E. Strom (OOPSLA ’94). *Comment*: idem, with a focus on protocols, and thus closer to the session–orientation of operations.