Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 182 (2007) 33–38
www.elsevier.com/locate/entcs

From Theory to Practice in Distributed Component Systems

Denis Caromel[1]

INRIA, University of Nice - Sophia Antipolis, I3S-CNRS, IUF, France

Abstract

This paper summarizes the keynote talk given at FACS'06, Formal Aspect of Component Systems, Prague, September 2006. The paper provides both an overview and a perspective of the papers cited in the reference. To achieve effective distributed components, we rely on an active object model, from which we build asynchronous and distributed components that feature various valuable properties.
We will emphasize how important it is to rely on a precise and formal programming model, and how practical component systems can benefits from theoretical inputs.

Keywords: Asynchrony, distributed components, determinism, Open Source, Grid computing

1 Introduction

This talk will start by presenting theoretical results on determinism for asynchronous distributed components. It will then show how to apply those results in a practical implementation, available as Open Source within the ObjectWeb Open Source community. Further, current work aiming at defining a joint European component model for Grid computing (GCM) will be summarized. Finally, it will conclude with challenges at hand with component systems, especially work related to capturing behavioral properties. Current work aiming at defining behavioural models and techniques for hierarchical components will be introduced.

Along the course of this presentation, we would like to demonstrate how important it is to rely both on practical and theoretical approaches in order to tackle the complexity of current large scale distributed systems. The second statement has more to do with a technical perspective: active objects provide a powerful and sound foundation for both understanding and programming distributed component systems.

[1] Email: Denis.Caromel@sophia.inria.fr

doi:10.1016/j.entcs.2006.12.042

2 Asynchronous Distributed Objects

In order to deal with components, a precise and comprehensive programming model is needed to adequately build primitive programs to be used as building blocks at composition time. The papers [6,7] define an object-oriented programming model for concurrent, parallel, and distributed systems. We summarize here the main key features:

- *asynchronous calls*, for the sake of hiding latency and decoupling client-server interactions;
- *first-class futures*, for the sake of passing the results of asynchronous calls to other distributed objects without forcing unnecessary synchronizations, also avoiding deadlocks – futures are indeed single assignment variables;
- *wait-by-necessity*, for the sake of using, where possible, dataflow re-synchronizations of parallel entities;
- *collective synchronization operations*, for the sake of manipulating synchronizations as first-class entities, *e.g.* blocking on the availability of all futures in a vector;
- *service primitives*, for the sake of programming in a flexible manner the inner synchronization of activities; and,
- *typed asynchronous groups*, for the sake of enabling asynchronous remote method invocations on group of entities, also a mechanism facilitating parallel component invocation.

Somehow, the features above propose a disciplined way to manage parallelism, and many users' operations are achieved in a parallel way without the burden to explicitly build complex synchronizations. Nevertheless, the programming model features a few fundamental properties:

- no interleaving within user code;
- no sharing of objects between concurrent threads; and,
- no unstructured callbacks.

The requirement – to have parallelism achieve users' operations – seems to be conflicting with the property of not having parallelism-enabling code interleaved within user code. Indeed, parallelism usually leads to interleaving of actions when conducted within a single address space. However, we rely here on the design and implementation of parallel operations within the middleware that have no consequences, whatsoever, for the user. This parallelism is risk-free, intrinsically acting towards confluence, because it does not produce any observable interleaving. Such harmless optimizations are indeed located at various places within ProActive's implementation, *e.g.* group communications, and future updates with automatic continuation. They are increasingly becoming more important with the advent of multi-core processors.

3 Calculus: Asynchronous Sequential Processes (ASP)

The ASP calculus provides a generalization of the ProActive programming model. It relaxes a few implementation decisions, and provides understanding and proofs of confluence and determinacy of asynchronous distributed systems.

Here are the main properties that were revealed by the ASP formal model:

- future updates can occur at any time, in any order, as such the delivery of replies can be implemented by numerous strategies, in any order;
- asynchronous FIFO point-to-point is sufficient for requests; and,
- the execution of a system is characterized by the order of request senders.

Those properties are further used in order to characterize several sets of deterministic types of programs:

- determinacy of programs based on a dynamic property (DON); and,
- determinacy of programs communicating over trees.

Let us also mention that some programs have become deterministic with Atomic Group Communications.

Overall, ASP provides a framework for understanding asynchronous distributed objects, the various potential implementation strategies, respecting strong properties of confluence and determinacy.

The difficulty is to statically approximate activities, method calls and potential services. Shifting to components will provide a statically defined topology.

4 Components

I would like to define a component in a broad sense as:

> *A software module, with a standardized description of what it needs and provides, its accepted parameters for configuration, and to be manipulated by tools for Composition and Deployment.*

In the framework of the GCM (Grid Component Model) as defined in [4] and implemented in ProActive, the components depict the following characteristics:

- Primitive Components featuring server and client interfaces;
- Composite Components, allowing the hierarchical composition of primitive and composite components to build large and structured configurations;
- Interface specifications to external languages such as: Java Interface, C++ .h, Corba IDL, and WSDL;
- Specification of Grid aspects such as: Parallelism, Distribution, Virtual Nodes, Performance Needs, and QoS;
- Multicast and Gathercast Interfaces to manipulate parallel behaviours at the level of interface specification rather than hidden in the code;
- Component Controllers, *i.e.* consider a controller as a sub-component, to provide

dynamic behaviour of the component control; and,

- Autonomic Components, the ability for a component to adapt to situations without relying on the outside.

Moreover, the GCM favors *asynchronous method calls*. By default, communications to the server interfaces are supposed to be non-blocking, as proposed in the ProActive implementation. Even in the case of methods returning non-void values, the caller is not supposed to be blocked during the method service. Together with the first-class futures as described above in the framework of ProActive and ASP, it provides the capacity to build both *structured and asynchronous* component configurations.

Using the theoretical ASP properties, it will become possible to identify deterministic components in practice, first based on the detection of deterministic primitive components, further by the characterization of deterministic composition of primitive components. Overall, components provide a convenient abstraction for statically ensuring determinism.

As identified before, one of the difficulties with deterministic distributed programs was to statically approximate activities, topologies, distributed method calls, and services. Moving to configurations defined through components, and providing a statically defined topology, makes system analysis a lot easier, and very practical. Indeed, the programmer usually has a clear idea about his program topology, therefore trying to automatically discover it makes things unnecessarily complex; it is also undecidable. Instead of using the topology provided by the programmer, we take a stand to help the programmer achieve what he is willing to do, rather than trying to tell him from scratch the properties of his programs.

5 Vercors: Model-Checking Distributed Objects and Components

The effort described in [5] aims at *model-checking* asynchronous distributed systems following the semantics of asynchronous active objects. That includes dealing with components as defined in the section above.

The approach pursued for model-checking is a compositional modeling of primitive components using Parameterized Labeled Transition Systems (pLTS). In order to achieve large-scale distributed verification of applications in an effective manner, we rely on the hierarchical approach, pLTS composed with pNET (parallel Networks).

In a nutshell, the approach adopted to achieve successful model checking is rather practical. We try as much as possible to use several sources of information (program source, architecture described through ADL, modeling using UML diagrams, and defining primitive component behaviour using State Machine Diagrams. In the future, we could envision using standardized code annotations, provided by the user, to pass key information to the model-checker with the increased probability to maintain them coherently, the source and the meta-information being both in the

same file.

The key features and properties coming from the active object model and ASP being used in the model-checking verifications seem to be:

- wait-by-necessity;
- future update can occur anytime with no consequences;
- there is no sharing between active objects;
- no user-level, code-level concurrency and parallelism; and,
- the behaviour of programs is insensitive to distribution/location of activities within address spaces (JVMs).

6 Conclusion: practice in the ProActive middleware

The ProActive middleware proposes a full-fledged environment with the programming of primitive code, the composition of such codes into composite components, the deployment on various practical infrastructures, and Graphical User Interface (Eclipse Plugin) to help programming, debugging and testing.

One of ProActive's key features is the combination of systematic asynchronous method-call, together with wait-by-necessity and first-class futures. At the level of components, it translates into the strong properties of large composition of composites not being blocked by synchronous calls.

Within the GCM, collective operations, previously achieved at the level of objects, are being abstracted into elements of the interface. This shift first represents an achievement in terms of readability, and reuse. Second, functional methods can be used in various contexts, standard non-collective code and at the same time in powerful group interactions. Moreover, it also achieves an important increase in the level of abstraction used by the programmer: interfaces versus the old API style for controlling parallelism, multicasting and synchronizations. Finally, it permits typing of collective behaviour.

From an historical stand, with a hierarchy of *modules* then *objects* then *components*, components could be viewed as moving backwards in programming evolution. We are moving to a more static topology, while we have shifted from module (static assembly) to objects where the inter-connection between pieces of code is often purely dynamic. With components, the interconnection is static, and can only move back to dynamicity using *controllers* at execution, like binding controllers. In other words, only some specific entities of the architecture authorize the introduction of dynamicity. Somehow, components can be viewed as *dynamicity under control !*

Why does it scale? Thanks to a few key features like typed, asynchronous (connection-less) communications – unified RMI+JMS, with messages rather than long-living interactions.

Why does it compose? First, because it scales! Indeed one would not be able to scale up to very large component configurations without the benefits of asynchronous method invocations. Second, the model composes because of its typed

nature: remote method invocations typed with interfaces. One would not be able to check large systems without some of the guaranties given by a static type system. The absence of unstructured *call-backs* and *ports* makes a tremendous difference with respect to verifying a component system.

As much as possible, we try to use static relations provided by component configurations, avoiding a great deal of static analysis. We believe dynamicity has to be mastered in the future with appropriate controllers, such as binding controllers. As an envisioned development, specific properties demonstrated on such controllers can be further used in a dynamically evolving system to prove global properties needed in complex, adaptive reconfigurations.

To conclude, the strategy embraced for verifying real applications – as many of us pursue in the community of *Formal Aspects of Components Systems (FACS)* – is to let the user provide as much information as possible rather than trying to discover non-decidable facts about the programs. We believe that it is impossible to *tell* the user what he is doing, but instead it is possible to *verify* automatically on his behalf what he thinks he is doing. *Rather checking than guessing what the user is doing*, that could summarize our current approach.

Acknowledgement

The ASP work is a joint work with Ludovic Henrio, and the Model Checking aspects are conducted under the supervision of Eric Madelaine. I am also most grateful to the entire OASIS team whose work has been decisive in the development of the ProActive middleware.

References

[1] Object ProActive, http://ProActive.ObjectWeb.org/

[2] Caromel D., Henrio L., Serpette B., "Asynchronous and Deterministic Objects", pp. 123–134 in Proceedings of the POPL'04, 31st ACM Symposium on Principles of Programming Languages, ACM Press, POPL, 2004.

[3] Caromel D., Henrio L., "A Theory of Distributed Objects", Springer-Verlag, 2005, 378 pages, hardcover, ISBN 3-540-20866-6.

[4] GCM: Grid Component Model, NoE CoreGrid deliverable, 2006 Towards a European Standard Component Model for Grid Computing.

[5] Behavioural Models for Hierarchical Components, T. Barros and L. Henrio and E. Madelaine, Model Checking Software, 12th International SPIN Workshop, Springer Verlag, 2005, Aug., San Francisco, pp. 154–168, LNCS.

[6] Towards a Method of Object-Oriented Concurrent Programming, D. Caromel, pp. 90-102, in CACM, Communications of the ACM, Volume 36, Number 9, September 1993.

[7] Programming, Composing, Deploying for the Grid, Baude F., Baduel L., Caromel D., Contes A., Huet F., Morel M. and Quilici R., in "GRID COMPUTING: Software Environments and Tools", Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, January 2006.