Agents Everywhere, All the Time [1]

Oscar Nierstrasz, Jean-Guy Schneider, Franz Achermann
Software Composition Group, IAM, Universität Bern
Neubrückstrasse 10, CH-3012 Berne, Switzerland
Tel: +41 (31) 631.4618. Fax: +41 (31) 631.3965.
{oscar,schneidr,acherman}@iam.unibe.ch. www.iam.unibe.ch/~scg.

1. ECOOP 2000 Workshops #5 (Component-Oriented Programming) and #22 (Pervasive Component Systems).

Abstract

Moore's Law is pushing us inevitably towards a world of pervasive, wireless, spontaneously networked computing devices. Whatever these devices do, they will have to talk to and negotiate with one another, and so software agents will have to represent them. Whereas conventional services on intranets will continue to be distributed using established middleware standards, internet services are being built on top of http, wap or other protocols, and exchange information in HTML, XML, and just about anything that can be wrapped as a MIME type or streamed. This situation leads us to three software problems: (i) How can we simplify the task of programming these agents? (i.e., Java is not enough), (ii) How can agents interact and interoperate in an open, evolving network environment? (i.e., XML is not enough), (iii) How can we reason about the services that agents provide and use? (i.e., IDL is not enough). We discuss these questions in the context of our work on Piccola, a small composition language, and outline ongoing and further research.

1. Why Agents?

Plummeting hardware costs, advances in miniaturization, durability, reliability, network technology, and battery life are enabling a world of pervasive computing. Although no one can reliably foresee even a fraction of the new applications for this technology, it is already clear that more and more computing power will appear in wireless, mobile devices, and that these devices will have to spontaneously interact in constantly changing environments.

As a consequence, we can expect to see the computing world divide into two rather different environments. The first is the conventional world of databases, servers, applications, and statically networked computers. Software services and components will continue to be implemented with conventional technology, and software systems will be composed as they are today. The second world will be a wireless one of spontaneous networking and constantly changing internet services. The clients of these services will include not only humans, but also other software systems, as service providers strive to reach the broadest customer base possible. These worlds will increasingly require agents to

- represent both human and software customers and providers (e.g., meta search engines),
- wrap legacy software systems (e.g., database systems),
- represent roaming devices in changing environments (e.g., mobile phones),
- encapsulate coordination protocols (e.g., Jini [4]),
- coordinate other agents (e.g., distributed agenda management).

We will not worry too much about developing a precise definition of software "agents", or even about comparisons with older definitions, except to say that they are software programs that can act autonomously on behalf of a human, an organization, or another software or hardware system [11]. We do not require that agents be mobile or intelligent. On the other hand, we are interested in the characteristics that these agents will exhibit:

- agents represent and wrap services for potentially remote hardware or software systems,
- agents coordinate and manage multiple incoming or outgoing requests,
- agents must adapt to changing network environments,
- agents may need to negotiate quality-of-service or other non-functional properties with other agents.

The rest of this paper is organized as follows: in section 2, we introduce important aspects related to the composition of distributed agents. In section 3, the need for a data model suitable for inter-agent communication and collaboration is discussed. In section 4, we propose notions to support reasoning about agent systems in order to build reliable applications. Finally, in section 5, we summarize the main observations and present directions for future research.

2. Composing Agents

Programming concurrent and distributed applications is not easy, so there is no reason to expect that programming agents will be. We can expect, however, to take advantage of some characteristics of agents to simplify the task.

As we have seen, we can expect agents to be mainly concerned with wrapping, composing, and coordinating existing services, not with implementing new basic services. Therefore, a language or tool for implementing agents should concentrate on supporting high-level abstractions that will simplify these tasks.

Java has been touted as the language of the internet, and it provides many features that make it easier to develop internet applications. Nevertheless, Java is still just a conventional programming language, better suited to implementing components than to composing them [8]. Furthermore, Java is not well-suited to defining abstractions which are not objects (such as coordination abstractions, generic wrappers, and synchronization policies).

We have argued elsewhere that component-based software development can be summarized as "Applications = Components + Scripts" [1][3][14]. In this view, conventional programming languages will be used to implement basic services and components, but a *composition language* will be more suitable for the task of composing, or "scripting" components. Furthermore, a composition language must pay special attention to the problems of "glue" (i.e., overcoming compositional mismatch), coordination (i.e., coordinating concurrent and distributed components and requests), and expressing various architectural styles.

This last point is actually the key both to composing software components and to composing agents. An "architectural style" [16] formalizes the components, connectors and composition rules for an application domain. In essence, an architectural style allows you to raise the level of abstraction from low-level "wiring" of component interfaces to composition using domain specific abstractions. Pipes and filters are the classical example, but there are countless other domains where architectural styles can help to raise component composition to a more declarative level [5]. Consider, for example, low-level wiring of events and actions as opposed to graphical composition of GUI widgets or rule-based specifications of trigger conditions and actions.

Piccola is an experimental composition language that is suitable to specify systems as compositions of agents. It is a small language based on the notions of immutable *forms*, which encapsulate data and services, *agents*, which encapsulate behaviour, and *channels*, which are the media that agents use to communicate forms. The formal semantics of Piccola is based on the πL-calculus, a process calculus in which agents communicate forms (extensible records) rather than tuples [10]. Forms are not only used exchange data in agent communications, but also to model environments, extensible interfaces, components, and objects [15].

We have already demonstrated that Piccola is well-suited to expressing various architectural styles [1], and that its simple, uniform model allows it to be used to express various composition and coordination abstractions, such as synchronization policies, that are clumsy or impossible to express in conventional object-oriented languages.

We plan future experiments to use Piccola for visualizing agent systems and to build a composition environment on top of Piccola. Furthermore, we are in the process of extending the run-time system to cope with real distributed applications.

3. Communicating with other Agents

Although Piccola is nice for wrapping, gluing, and composing components in a closed environment, it still lacks the reflective features that would be needed to use it to program agents that could function effectively in a completely open, spontaneously networked environment. We are currently investigating which lightweight reflective mechanisms are needed to solve this problem.

In closed environments today, component interfaces are typically specified with IDL or Java, and clients communicate with them through CORBA, COM, or RMI. Internet services, on the other hand, are specified as HTML forms, return HTML, XML, or various MIME types, and clients communicate with them through http, ftp, wap, or any of a variety of other protocols.

It is clear that the world is moving towards XML as a medium for exchanging models between software systems, so we can expect that interoperable internet services will have to talk XML. But XML is more syntax than semantics, and it is not a full-fledged data model supporting a query language and data manipulation language. With XML, we risk returning to the days of hierarchical and network data models with procedural navigation through databases rather than declarative expression of views of data.

It is a fairly straightforward exercise to map XML to any kind of object model, and the forms exchanged by Piccola agents are no exception. Therefore, we imagine Piccola agents communicating with external services by means of XML. However, to query, manipulate, explore, and negotiate with the outside world, agents would need a declarative form data model.

Such a data model might resemble the relational model, with data being stored in bags of forms rather than sets of tuples (it is possible to adapt the operators of the relational al-

gebra to such a data model). The fundamental difference, however, is that the relational model is founded on a *closed world assumption*, whereas a data model for communicating agents must be based on open systems. Agents must be prepared to deal with new services and models for which they have not been preprogrammed. Therefore, a form data model must support some simple reflective features that allow agents to explore the schema to which a model conforms, or, better yet, to easily wrap models to conform to different schemas.

We are now exploring the definition of such a form data model, and experimenting with simple mechanisms to query, view, adapt, and transform form-based models.

4. Reasoning about Agent Composition

It is notoriously difficult to reason about software, and reasoning about open, concurrent, distributed agent software will not be any easier. The need for reasoning, however, will be even more acute, since agents will not be able to act autonomously if they cannot give and receive specific service guarantees.

Reasoning today is often limited to simple static type checking, but even this level of guarantee cannot be easily reconciled with an open agent world, since static type-checking requires advance knowledge of interfaces, and type inference typically requires global knowledge.

We can expect to move to a world in which "types" express other kinds of contracts than simple interface signatures, and the checks will be performed as much at run-time as at composition time. Therefore, we can see the need for:

Interfaces. Agents, like the components they represent, will provide and require interfaces, but run-time querying will become more important. Furthermore, interfaces will have to be versioned and assigned unique identifiers (similar to the global unique identifiers of COM [13], but perhaps these identifiers will be URLs), and wrapping services may be required to automatically adapt otherwise incompatible interfaces.

Quality of service. Some simple forms of quality-of-service negotiation are already commonplace in the domain of internet multimedia streaming. As ever wider ranges of devices and software agents become internet-aware, we can expect not only bandwidth and latency, but also other aspects of services, such as hard or soft real-time constraints, or platform dependencies, will become negotiable depending on the capabilities available to the client.

Protocols. Many kinds of services require clients to obey a simple kind of protocol, but flat interfaces tell us nothing about non-uniform service availability [12]. Many kinds of services require an initialization or authorization protocol to be completed before other actions may be performed. Furthermore, transaction services require a commit or abort from the client. Although we can expect most protocols to be quite simple and fall into easily recognizable classes, they must still be explicitly documented, statically checked, and dynamically validated.

Security & Authorization. Much has been made of "sandbox" techniques to protect servers from hostile agents, but what about protecting the agents themselves? Agents for roaming devices may easily find themselves in hostile environments. Rather than relying on ad hoc run-time methods to protect agents and their clients, it should be possible to decorate agent implementation code with explicit security assertions. Similar to protocols, a combination of static and run-time checks would be performed. It is unlikely that a single, internet-wide security model would be appropriate, since security requirements are known to vary considerably for different application domains. Therefore, agents must be able to adapt their security requirements to the capabilities provided by the environments they find themselves in.

Aliasing. In a certain sense, agents *are* aliases. Any situation where aliases cause problems in conventional software systems (deadlock detection, garbage collection, etc.) will certainly arise in agent systems as well. Techniques like *islands* [6] could well be reinterpreted from the perspective of agents. Again, we can expect that explicit assertions will help to limit and reason about aliasing.

Many important forms of reasoning can be reduced to the question "Can I get there from here?" Not only reasoning about aliasing, but also about security rights and capabilities or deadlock detection or prevention often takes this form. One can imagine an agent language with generic support for reasoning about reachability, which can then be adapted to specific problem domains.

Cost and Ownership. Reasoning about linear capabilities and ownership is typically not directly supported by programming languages, but must be programmed in an idiomatic way [9]. Agents, however, will often be responsible for various kinds of finite resources (not just money and credit, but also time, computing resources, and other physical and virtual resources), and must be able to reliably manage these resources, possibly in the presence of unreliable networks. Again, explicit assertions about transfer of ownership of linear capabilities would be of immense help if supported by static and dynamic enforcement procedures [7].

This list is not intended to be exhaustive, but rather to illustrate the need for "type systems" that allow us to express contracts required and provided by agents that go beyond simple interface signatures. These type systems should allow us to statically check the sanity of our agents, but will typically also require some run-time support to validate that advertised contracts are actually respected.

5. The Upshot

We claim that one of the key challenges in developing component-based systems of the future will *not* be programming of individual components, but rather programming the agents that represent, wrap, coordinate, and compose them. The other key challenge, not discussed here, is how to *migrate* existing systems to component-based frameworks.

We argue that the challenge of agent coordination must be addressed by new programming paradigms, and we propose "Applications = Components + Scripts" as such a paradigm. Furthermore, we offer Piccola, a small composition language, as an effort in that direction.

We are continuing to experiment with Piccola, by using it as a medium for expressing compositional styles for various application domains [2]. Furthermore, we are experimenting with the development of a *form data model* that will enable agents in open networks to exchange, query, manipulate, and adapt models expressed using interchange standards like XML. Finally, we are investigating ways for reasoning about the *contracts* that agents require and provide on behalf of their clients.

Acknowledgements

We would like to all members of the Software Composition Group for their support of this work, especially Markus Lumpe for helpful comments on an earlier draft. This work has been supported by the Swiss National Science Foundation under Project No. 20-53711.98, "A framework approach to composing heterogeneous applications", and the Swiss Federal Office for Education and Science under Project BBW Nr 96.00335-1, within the Esprit Working Group 24512: "COORDINA: Coordination Models and Languages."

References

[1] Franz Achermann and Oscar Nierstrasz, "Applications = Components + Scripts — A tour of Piccola," *Software Architectures and Component Technology*, Mehmet Aksit (Ed.), Kluwer, 2000, to appear.

[2] Franz Achermann and Oscar Nierstrasz, "Explicit Namespaces," Proceedings of JMLC 2000, 2000, to appear.

[3] Franz Achermann, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, "Piccola - a Small Composition Language," *Formal Methods for Distributed Processing, an Object Oriented Approach*, Howard Bowman and John Derrick. (Eds.), Cambridge University Press., 2000, to appear.

[4] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo and Ann Wollrath, *The Jini Specification*, Addison-Wesley, 1999.

[5] Len Bass, Paul Clements and Rick Kazman, *Software Architecture in Practice*, Addison-
Wesley, 1998

[6] John Hogg, "Islands: Aliasing Protection in Object-Oriented Languages," *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, number 11, November 1991, pp. 271-285.

[7] Naoki Kobayashi, Benjamin C. Pierce and David N. Turner, "Linearity and the π-Calculus", *Proceedings of Principles of Programming Languages (POPL'96)*, ACM Press, 1996.

[8] Danny B. Lange and Mitsuru Oshima, "Mobile Agents with Java: The Aglet API," *World Wide Web Journal*, 1998.

[9] Doug Lea, *Concurrent Programming in Java — Design principles and Patterns*, Addison-Wesley, The Java Series, 1996.

[10] Markus Lumpe, "A Pi-Calculus Based Approach to Software Composition," Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[11] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran and J. White, "MASIF, The OMG Mobile Agent System Interoperability Facility," *Proceedings of Mobile Agents '98*, 1998.

[12] Oscar Nierstrasz, "Regular Types for Active Objects," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pp. 99-121, Prentice Hall, 1995.

[13] Dale Rogerson, *Inside COM: Microsoft's Component Object Model*, Microsoft Press, 1997.

[14] Jean-Guy Schneider and Oscar Nierstrasz, "Components, Scripts and Glue," *Software Architectures — Advances and Applications*, Leonor Barroca, Jon Hall and Patrick Hall (Eds.), pp. 13-25, Springer, 1999.

[15] Jean-Guy Schneider, "Components, Scripts, and Glue: A conceptual framework for software composition," Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[16] Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.