A Formal Approach for Interoperability between Mobile Agent Systems and Component Based Architectures

Ulrich Pinsdorf
Fraunhofer Institute for Computer Graphics
Fraunhoferstraße 5, 64283 Darmstadt, Germany
ulrich.pinsdorf@igd.fraunhofer.de

Abstract

We present an algorithm that allows to cut a component from a component based system and transplant it into a mobile agent system. We refer to this process as grafting. *The overall goal is to run grafted components in a new execution environment. The approach is generic as it brings the target system into a position to execute* any *instance of the grafted component type. Moreover it applies for any two systems, not only for component based and agent systems. The approach was tested by grafting OSGi bundles into the mobile agent system* SEMOA.

1 Introduction

Mobile agents can be seen as tuple of program code, data and execution state, which migrates from one agent server – an agent execution environment – to another. Accessing resources locally enables an agent to collect, process and publish data efficiently within a network. Mobile agents push the flexibility of distributed systems to their limits since not only computations are distributed dynamically, but also the code that performs them. They may roam a network, seek information, and carry out tasks on behalf of their senders autonomously. Hence, mobile agents offer great benefits to applications in networks by adding client-side intelligence and functionality to server-side services.

Nearly each mobile agent system is founded on research activities with a specific focus and therefor follows a different design goal. Hence, each mobile agent system offers different strengths, such as security, scalability, enhanced agent behavior, efficient migration, etc. Unfortunately mobile agent technology lacks interoperability between different systems, which prevents the technology from reaching "critical mass´´ for widespread application. We think that opening mobile agent systems both for other mobile agent systems as well as for any component (such as Java Enterprise Beans, OSGi bundles, etc.) will push them into position to reach a higher impact and to bring their strengths on the market.

We define *run-time interoperability* as the ability of a (mobile agent) system to start any software component of a different component-based system and act as full replacement for the component's original run-time environment. That means that a software component running in an interoperable system may behave as usual without any modifications.

In this paper we show a way to take a software component from a component based system and transplant it into an agent system. The process includes modifications at the target agent system which eables it henceforth to execute any component of the specific type. We refer to the transplanting process as *grafting*. The component based system may be a second agent system just as a general component-based system. Altough our appoach is stated in a formal manner, it is derived from practical experience [22]. The ideas in this paper may not only be applied to mobile agent systems but to any two component based system, which depends on dynamically loaded code.

In section 2 we give an overview on related work both of mobile agent interoperability and the related area of object graph analysis. We present object graph analysis in section 3. This is only a short summary of the underlying theory, but enough to understand the formal description of the grafting process. Section 4 formally describes the grafting process which allows the transplantation of components. In section 5 we present experiences we made with integration of OSGi bundles within the agent platform SEMOA using the formal algorithm. Finally, section 6 concludes the approach and the lessons learned.

2 Related Work

In this section we give an overview on prior art. We focus on the areas of mobile agent interoperablity and on object

graph analysis. The latter is a technique for reverse engineering of object-oriented systems.

2.1 Mobile agent interoperability

In the area of mobile agent interoperability only a small number efforts have taken place so far. A well known approach is the MASIF proposal [19] by Milojicic et al which suggests a standardization for an agent transport protocol. Although the publication is one of the earliest and best known in this area the document did not have much public scrutiny yet. The platforms SOMA [33] and Grasshopper [15] are the only known mobile agent plattforms that implement the MASIF proposal.

FIPA (Foundation for Intelligent Physical Agents) is also active in the standardization of agent mobility issues [8], but this particular thread of FIPA's work focuses on a high level of abstraction, e.g. communication protocols, ontologiesm etc. However, only the FIPA standardization on agent communication issues [9] found broad attention.

Bellavista et al, the developers of the SOMA mobile agent system, suggest an approach on mobile agent interoperability [1, 2] which is based on the CORBA IIOP protocol [6]. SOMA platforms register themselves as CORBA servers and offer services to CORBA clients by means of IIOP protocol. This approach relies on communication does not establish run-time interoperability.

So beside our own publication [22] we are aware of only two publications which focus on the aspect of executing agents in foreign environments. Magnin et al. suggest an standardized adoption layer between agent and agent system [18]. This binds developers of interoperable agents to the API of the adoption layer. They can neither take advantages from the agent's programming model, nor from the server environment. The publication of Grimstrup et al [13] achieved basic interoperability between four agent systems by means of a common interoperability API an a number of translators from and to each native platform API. This in fact bears the same problem as Magnins approach.

In conclusion it is fair to say that existing standardization efforts – if available – have not yet shown to be *effective* to provide actual interoperability among systems for mobile agents. Hence, rather than following the top down approach to interoperability by means of standards, we chose to take a bottom up approach based on voluntary interoperability with other systems for mobile agents. We designed the system SEMOA [23, 10] in a way that it facilitates the task to provide true interoperability with other (agent) systems.

2.2 Object graph analysis

Our work also touches the area of object-oriented software engineering, and object-oriented reverse engineering. In these areas a large number of publications are available, [4, 16, 25] give a good overview.

In the area of object-oriented analysis a publication on object graph analysis by Spiegel [26, 27] fits to our approach very well. He uses object graphs for separation and distribution of software.

A number of projects have the goal to distribute monolithic software programs dynamically. The foremost prominent representative are the projects Doorastha [7], J-Orchestra [32], and JavaParty [21]. All of which analyze given code, part it, and distribute it to different hosts. This allows parallel computing and more efficient resources usage.

3 Object Graphs

The approach presented in this paper bases on static object graphs. An object graph is a formal representation of objects that are expected to exist during the run-time of a program[1]. As we will see, the object graphs and their according class graphs enable us to build a formal model to establish interoperability on implementation level between two systems that are mutual unknown to each other. The reason, why we start the code inspection with objet graphs and not with class graphs is that mobile agent system make intensivly use of dynamically loaded code. Thatfor we have to consider the estimated object graph and derive the class graph from that.

The nodes of a object graph represent the objects. Between objects one can distinguish three different kinds of edges: creation edges, reference edges and usage edges. Between instantiated objects we can distinguish between different relations:

- an object *a creates* an object *b* if the the statement which allocates *b* is executed in the context of object *a*;
- an object *a references* an object *b* if, at any time during execution, a reference to *b* appears in the context of *a* (either as a field, variable, or parameter, or as the actual value of an expression);
- an object *a uses* an object *b* if *a* invokes any methods of accesses any field of *b*.

We introduce a notation based on graphs in Table 1. It is an extention of the notation Spiegel gives in [26]. The object graph of an object oriented system consists of run-time objects (nodes) and their relations (edges). Applying

[1] We assume that this program was designed and implemented in an object-oriented programming language.

$\mathcal{G}$	Static object graph
Γ	Class graph
$\mathcal{O}, \mathcal{S}, \mathcal{T}$	Set of objects
C	Set of classes
$\mathcal{E}$	Set of edges between objects
E	Set of edges between classes
$a, b, c, \ldots$	Instantiated objects
$\alpha, \beta, \gamma, \ldots$	Object types (classes)
$\|\mathcal{O}\|, \|A\|$	Mightiness of the given set
(a, b)	Relation between objects a and b
(α, β)	Relation between types α and β
$\alpha = \tau[a]$	Type of a
$C = \tau[\mathcal{O}]$	Type of all elements in $\mathcal{O}$

Table 1. Summary of notation for static object graphs and static class graphs.

the notation, we could describe an object graph as:

$$\begin{aligned} \mathcal{G} &= \langle \mathcal{O}, \mathcal{E} \rangle \\ \mathcal{E} &= \mathcal{E}_c \cup \mathcal{E}_r \cup \mathcal{E}_u \end{aligned} \tag{1}$$

$\mathcal{O}$ is the set of objects that exists during run-time, and $\mathcal{E}_c$, $\mathcal{E}_r$, and $\mathcal{E}_u$ are *creation edges*, *reference edges*, and *usage edges*. We write (a, b) if an object a has a relation to an object b.

Since every object has a specific type there is a surjective mapping between an object graph G and a class graph Γ. A class graph consists of a set of object types/classes (nodes) with dependencies between them (edges). An object graph $\mathcal{G}$ has a corresponding class graph Γ:

$$\begin{aligned} \Gamma &= \langle C, E \rangle \\ E &= E_e \cup E_i \cup E_u \end{aligned} \tag{2}$$

C is the set of classes C. E_e, E_i, and E_u denote *inheritance edges*[2], *implementation edges*, and *usage edges*. The mapping between object graph G and class graph Γ is as follows:

$$\forall x : x \in \mathcal{O} \quad \Rightarrow \quad \exists\, \alpha : \alpha \in C \wedge \alpha = \tau[x] \tag{3}$$

$$\forall (a, b) : (a, b) \in \mathcal{E} \quad \Rightarrow \quad \exists\, (\alpha, \beta) : (\alpha, \beta) \in E \wedge \alpha = \tau[a] \wedge \beta = \tau[b] \tag{4}$$

In section 4 we use this theory for describing the grafting process. Finally, in section 5 we show how the formal approach had been applied for an adoption of OSGi bundles within the agent platform SEMOA.

4 Grafting Process

In this section we present an algorithm that allows to cut a component from an component based system and adopt it in an mobile agent system. We refer to this process as *grafting*, the component's home system is called *alien system* and its runtime environment *target system*. The overall goal is to execute grafted components in its new environment. The approach itself is generic as it brings the target system into a position to execute *any* component designed for the alien system. Moreover it applies for any two systems, not only for component based and agent systems. It is important to understand that we do not only a cutting but also an analyzing of the alien system. Since we are interested in the general interaction methods between an alien component and its environment, the grafting process introduced in this section will likely transplant more than just the component classes.

The general process consists of three steps:

1. analyzing the alien system,
2. cutting-out of the alien component, and
3. adoption within the target system.

We describe those three steps in the following sections. Let $\mathcal{G}$ be the object graph and Γ be the class graph of the alien system including the component. Further let α be the starting[3] class of an alien component which shall be grafted. Then a is the according starting object for one specific component instantiation. We write $a = \tau[\alpha]$.

4.1 First step: Analyzing

This is the most difficult step. In order to cut a out of $\mathcal{G}$ (and likewise α out of Γ) we have to determine a subgraph $\mathcal{G}' \subset \mathcal{G}$ with $\mathcal{G}' = \langle \mathcal{O}', \mathcal{E}' \rangle$, $\mathcal{E}' \subseteq \mathcal{E}$ and $\mathcal{O}' \subseteq \mathcal{O}$. In addition $\mathcal{G}'$ shall be bound to the following restrictions

1. $a \in O'$,
2. $\mathcal{O}'$ includes all objects which are essential for the grafted component's functionality, and
3. $|\mathcal{G}'|$ is minimal.

The latter condition guarantees an optimal cut-out with maximal re-using of component classes and a minimum number of references to be cut. In other words, the essential functionality shall be cut out whereas the supporting system shall be left behind.

For determine $\mathcal{G}'$ we calculate the transitive hull in $\mathcal{G}$ of all objects used by the alien component, starting at a. Figure 1 shows the algorithm for finding $\mathcal{G}'$.

After initialization of the sets $\mathcal{O}'$, $\mathcal{S}$ (stack), and $\mathcal{E}'$, we begin at object a by pushing it on top of the empty stack S.

[2]The index *e* stands for *extends*.

[3]The component itself may consist of a number of classes, but there is always one specific class that is used for instantiating or representing the component.

$\mathcal{S} \leftarrow \{a\}$
$\mathcal{O}' \leftarrow \emptyset$
$\mathcal{E}' \leftarrow \emptyset$
while $\mathcal{S} \neq \emptyset$ **do**
  $x \leftarrow \text{pop}(\mathcal{S})$
  $O' \leftarrow O' \cup x$
  foreach $(x, d) \in \mathcal{E}$ **do**
    if $d \notin \mathcal{O}' \wedge \tau[d] \notin T$ **then**
      push($\mathcal{S}$,v)
      push($\mathcal{E}$,(x, d))
    endif
  done
done
$\mathcal{G}' \leftarrow < \mathcal{O}', \mathcal{E}' >$

Figure 1. Search for the transitive hull of all necessary objects of a.

We check successively all relations starting at object called x which is the actual top of the stack. For each relation starting at x we check whether the destination object d is not already in the object collection $\mathcal{O}'$ *and* whether the class of d belongs to a set of *terminal classes* denoted as T. The first constraint helps to deal with circles in the object graph. The set of terminal classes T in the second constraint is predefined, and we describe them in detail in the next section. Whenever the algorithm encounters such a class it stops further inspection in this direction. Each object x that passes these constraints belong to the transitive hull and is pushed on the stack $\mathcal{S}$ for later inspection. We remember the inspected class in $\mathcal{O}'$ and the edge (x, d) in $\mathcal{E}'$. These steps are repeated as long as at least one class resides on the stack $\mathcal{S}$. After $\mathcal{S}$ is consumed $\mathcal{O}'$ holds all objects that are somewhat essential for a. $\mathcal{G}'$ is simply the graph resulting in the combination of objects nodes $\mathcal{O}'$ and their relations $\mathcal{E}'$.

This algorithm vitally depends on the set of terminal class types T since this set determines which objects are essential and eases an cutting-out. Thus the definition of terminal object types is one of the key points in our approach. We define[4] classes belonging to the following class types as *terminal classes*:

- **Interfaces classes** define the accessors to a class without defining its functionality. Interfaces represent the ideal terminal classes since the usage of an interface method is well-defined but not bound to an implementation.
- **Abstract classes** are un-instantiateable classes. They usually act as base classes which provide on one hand a common functionality, and on the other hand leave certain methods intentionally blank.
- **Standard classes** that constitute the extent of each programming language. Given that they are also available for the adapting platforms they are not essential for the agent.
- **Dynamic loaded classes** are commonly used for loading drivers at run-time. Usually it is hard to backtrace the object graph at dynamically loaded classes.
- **Isolated helper classes** without references to other classes $c \notin T$ can often be reused without modification.
- **Native anchored classes** are classes which depend on native code, e.g. has native methods.
- **Modules** typically use classes that resemble entry points to self-contained subsystems. Such modules need no special adaption and can be used as a whole. In other words, modules that can be treated as a black box and need not belong to the agent itself.

Whenever a class inside the transitive hull references a class belonging to a classes type above, we stop further inspection in this direction. This leads to a minimal object graph $\mathcal{G}'$.

$$\mathcal{O}' \subseteq \mathcal{O} \setminus \{x \in \mathcal{O} : \tau[x] \in T\} \tag{5}$$

Hence, the cardinality of T and $\mathcal{O}'$ depend from each other:

$$|\mathcal{O}'| \sim \frac{1}{|T|} \tag{6}$$

Thus a maximal $|T|$ concludes in a minimal $|\mathcal{O}'|$ and a minimal $|\mathcal{G}'|$ as demanded above. In contrast the set of terminals also specifies the objects that are *potentially* important for the functionality of the agent. If T contains too much elements it would exclude objects from $\mathcal{O}'$ that are necessary for a.

What we have so far is a set of objects that are related to the agent base class in a way that we have to import them along with the agent in oder to guarantee the agent's functionality.

4.2 Second step: Cutting-out

In order to make a "clean" cut-out, we need a cutting line in Γ including $\tau[a]$ and its essential helper classes. This line can be deduced from the object graph G', but first we need the class graph Γ.

The object graph $\mathcal{G}'$ can be transformed into the corresponding class graph Γ' as shown in equations (2) to (4). The algorithm's selection condition guarantees that

$$\Gamma \cap T = \emptyset \tag{7}$$

[4]For a language specific definition we recommend the specifications of the programming language, e.g. for Java [12].

In other words the cutting-line runs entirely along those references reaching outward from the grafted component to terminal classes:

$$\alpha, \beta \in \Gamma : \alpha \in \Gamma' \wedge \beta \in T \tag{8}$$

This has an important consequence for the grafting process. Since those classes – which is fact were only defined to be terminal – are initially designed to interact *within* the alien system, the target system has to satisfy their communicative needs. This leads us directly to the step of implanting.

4.3 Third step: Implanting

In order to execute the alien component properly, the target system has to act as adequate replacement for the remaining alien system $\Gamma \setminus \Gamma'$. More precisely, the target system has to intercept any outgoing communication of alien classes, generate an answer according to the target systems internal management, and pass it back to the alien component.

This leads to the advisory that terminal classes should be connected by means of intermediate classes which ensure the communication between both different systems. These intermediate classes could also perform a translation of parameters or break a single method call into a number of calls for the target system and vice versa.

Efforts to realize these intermediates are very manageable since they simply invoke functionality of the target system (e.g. access to data sources, component management, communication, etc.) which are usually well-known by the integrator.

We give two examples on how to realize intermediate classes. First, the most common type of terminal classes is the interface, which defines a well-defined functionality but do not provide an implementation. Hence, the adaption environment could implement any interface in Γ' and deliver the expected functionality backed by the target system. Second, inheritance from abstract classes is a way to integrate this type of terminal classes within the target system. Depending on the actual situation adaption of other terminal class types is possible in a similar way.

An additional issue is that the target system has to supply a way to distinguish the different types of supported components – their native component type as well as a number of supported alien systems – and accordingly set up different types of adaption environments. Each of those environments has to satisfy the needs of one specific component type. Or in other words, each adaption environment fakes a specific alien system. In a earlier publication [22] we coined the term *lifecycle*[5] for those environments and suggested a pattern for handling different kind of lifecycles. When a platform supports more than one type of adaption it becomes a problem to decide which kind of lifecycle to choose for an actual starting-up component. Thus we suggest the *lifecycle registry* which holds all types of supported lifecycles. Once, the component is admitted to the system it is passed to a lifecycle registry, which passes it subsequently to all registered *lifecycle factories* until a factory signals that it is willing to handle the agent's class. This factory (see also [11] for an introduction to the factory pattern) generates a *lifecycle instance* that can handle the component, and wraps around it. The lifecycle instance translates between SEMOA's native lifecycle and the lifecycle of the alien system. In particular, it instantiates all necessary components that make the component believe that it is running on its native system.

5 Experiences

The grafting process had been successfully tested with a number of alien agent system. SEMOA provides a successful integration for Aglets [17], Jade [3], and Tracy [5] agents. After that we turned to general component-based systems such as OSGi.

OSGi [31, 20] stands for *Open Standard Gateway Initiative* and is a well-established standard for component based systems in Java [12, 29, 30]. The consortium consists of 40 international members including Sun Microsystems, IBM, Motorola, Nokia, Toshiba, Deutsche Telekom, and others. Its intention is an open service platform for home automation and in-car systems. The alliance is open for new members and contributors.

The standard itself consists of a document part which describes the requirements for an OSGi-compatible service platform, and a software part providing a large framework of Java interfaces. In OSGi terms, independent components are called *bundles*, which come along as JAR files [28] and have to implement a small subset of OSGi interfaces. Each bundle provides a distinct functionality and may import other bundles for assistance. After startup of a bundle within its native service platform a call-back hook is set, that allows interaction with the surrounding system, again by means of interfaces. The service platform realizes the bundle management and enables requesting, importing and interaction between bundles. There are a number of service platform implementations available on the Internet. A well-known project we used for our adaption is the Open Service Container Architecture (OSCAR) [14].

We chose for our concrete analysis a standard bundle shipped with OSCAR (shell bundle). The grafting process started at the bundles activator class. Bundle activators can

[5]The term lifecycle was chosen, because they essentially establish a correspondence between the lifecycles of the alien system and the native lifecycle.

be considered as entry point for bundles, for they are the first to start and invoke all further actions. From here we determined all essential objects by calculating the transitive hull as shown above. The consequent use of interfaces defined by the OSGi standard emerged as ideal prerequisite for the grafting process, since interfaces are defined as terminal class type. The set of essential classes was small and fit to the set of classes contained in the bundle's JAR file. This is an indices for the excellent separation of bundle and service platform in OSGi. After cutting-out we implemented helper classes which provide the necessary interfaces and act as intermediate between SEMOA and OSGi. During our recent grafting processes with different agent systems we gained some experience and developed an interoperability layer which eases interaction of those intermediate lifecycle classes in SEMOA[22].

The main challenge was the integration of the expected bundle management within SEMOA's native component management. SEMOA uses for this sake a global hierarchical namespace, where all agents, services, transport handlers, etc. are registered under a unique path (refer [24] for more details). Bundles where simply registered within this namespace and could be looked easily up by any other OSGi lifecycle. Furthermore a number of small functionalities had to be implemented, e.g. a dictionary for property values or a mechanism for event dispatching. We tested our implementation with a number of free OSGi components and found all requirements fulfilled. The SeMoA platform is now capable to execute any OSGi bundle.

Our experiments showed that the formal approach proved as applicable for a real grafting process. However, a well designed and documented alien system helps a lot to achieve interoperability in a short amount of time. Especially the well documented semantic of interface methods were of great help. The grafting process was performed manually, without any implementation of the presented algorithm. Learning from these experience we think that a semi-automatic cutting-out is possible. The implanting step relies on programming skill and experience in system designed and will stay an act of creativity, which probably prevents it from automation.

6 Conclusions

In this paper, we presented a formal approach for runtime interoperability between mobile agent systems and component based systems. We described a process for grafting components taken from one system into a second system. This is based on voluntary interoperability between selected agent systems, rather than a top-down approach driven by standards. The approach itself is generic and may be applied to any kind of object-oriented software system.

In particular, we presented how static object graph analysis helps to create the estimated run-time object graph. We suggested an algorithm for separating essential from non-essential objects starting at the component's main object. The determined border line can be applied to the corresponding class graph, which helps to cut out the component from its surrounding system. Implanting the component within an agent system is possible by taking advantage from intermediate classes. They manage the translation of method calls and lifecycle states between component and the new hosting system.

Once the integration was achieved for a particular component, the target system is able to execute any other component of the specific type. Another important key feature of this approach is that the component may profit from the strengths of its hosting system and vice versa. E.g., when the target platform provides migration support or special security features, the grafted component type may take advantage from that. This allows the design of very flexible systems by bringing together the host's and component's benefits.

In the course of pursuing interoperability between different mobile agent systems we gained considerable insight both in the particularities of Java as well as in the dos and don'ts of component-based system design in general and mobile agent based system design in particular.

At the time of writing, our work is far from complete, yet we can already demonstrate a successful integration of Aglets, Jade, and Tracy agents. OSGi is the first component framework we have adopted. Currently we examine possibilities for a semi-automate integration of other component based systems.

References

[1] P. Bellavista, Antonio Corradi, and Cesare Stefanelli. CORBA solutions for interoperability in mobile agent environments. In *Proceedings of the 2nd International Symposium on Distributed Objects & Applications (DOA'00)*, pages 283–292, Antwerp, Belgium, September 21-23 2000. IEEE Computer Society Press. Available at URL `http://www.computer.org/proceedings/doa/0819/08190283abs.htm`.

[2] P. Bellavista, A. Corradi, and C. Stefanelli. Protection and interoperability for mobile agents: a secure and open programming environments. In *IEICE Transactions on Communication, Special Issue on Autonomous and Decentralized Systems*, volume E83-B, pages 961–972. The Institue of Electronics, Information and Communication Engineers, May 2000. Available at URL.

[3] F. Bellifemine, A. Poggi, and G. Rimassa. Jade programmers guide, June 2000. Available at URL `http://sharon.cselt.it/projects/jade`.

[4] G. Booch. *Object-Oriented Analysis and Design with Application*. Addison-Wesley Publishing Co., 2nd edition, September 1993. ISBN 0805353402.

[5] P. Braun, C. Erfurth, and W. R. Rossak. An introduction to the Tracy mobile agent system. Technical Report No. 2000/24, Friedrich Schiller University of Jena, Computer Science Department, September 2000. Available at URL ftp://ftp.minet.uni-jena.de/ips/braun/bericht-00-24.pdf.

[6] CORBA 2.6 specification. Technical Report formal/20011235, Object Management Group, 2001. Available at URL http://www.omg.org.

[7] M. Dahm. The Doorastha system. Technical Report B-1-2000, Institut für Informatik, Freie Universität Berlin, Takusstraße 9, 14195 Berlin, Germany, May 2000. Available at URL http://www.inf.fu-berlin.de/~dahm/doorastha/.

[8] FIPA agent management support for mobility specification. FIPA document PC00087A, Foundation for Intelligent Physical Agents, Jun 2000. Available from URL http://www.fipa.org/specs/00087/.

[9] FIPA ACL message structure specification. FIPA document XC00061E, Foundation for Intelligent Physical Agents, Aug 2001. Available from URL http://www.fipa.org/specs/00061/.

[10] Fraunhofer-IGD. SeMoA – Secure Mobile Agents Project. Website. Available at URL http://www.semoa.org/.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns*. Addison Wesley Longman Publishing Co., December 1994. ISBN 0201633612.

[12] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996. ISBN 0-201-63451-1.

[13] A. Grimstrup, R. S. Gray, D. Kotz, T. Cowin, G. Hill, N. Suri, D. Chaćon, and M. Hofmann. Write once, move anywhere: Toward dynamic interoperability of mobile agent systems. Technical Report TR2001-411, Dartmouth College, Compzuter Science, Hanover/NH, USA, July 2001. Available at URL http://www.cs.dartmouth.edu/reports/TR2001-411/.

[14] R. S. Hall. OSCAR – open service container architecture. Website. Available at URL http://oscar-osgi.sourceforge.net/.

[15] IKV++ Technologies AG. Grasshopper2. Website. Available at URL http://www.grasshopper.de/.

[16] I. Jacobson, G. Booch, and J. Rambaugh. *The Objectory Software Developement Process*. Addison Wesley Longman, 1998.

[17] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents With Aglets*. Peachpit Press, September 1998. ISBN 0201325829.

[18] L. Magnin, V. T. Pham, A. Dury, N. Besson, and A. Thiefaine. Our GUEST agents are welcome to your agent platforms. Submitted for publication.

[19] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Omo, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF – The OMG Mobile Agent System Interoperability Facility. In K. Rothermel and F. Hohl, editors, *Proceedings of the Second International Workshop on Mobile Agents (MA '98)*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67. Springer Verlag, Berlin Heidelberg, September 1998. The MASIF specification is available at URL http://www.fokus.gmd.de/research/cc/ecco/masif/doc/97-10-05.pdf.

[20] OSGi Alliance – home. Website. available at URL http://www.osgi.org/.

[21] M. Phillippsen and M. Zenger. JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[22] U. Pinsdorf and V. Roth. Mobile Agent Interoperability Patterns and Practice. In *Proceedings of Ninth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2002)*, pages 238–244, University of Lund, Lund, Sweden, April 2002. Institute of Electrical and Electronics Engineers, IEEE Computer Society Press. ISBN 0-7695-1549-5.

[23] V. Roth and M. Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pages 435–442, Dallas, Texas, U.S.A., March 2001. IEEE Computer Society. ISBN 0-7695-1065-5.

[24] V. Roth, U. Pinsdorf, J. Peters, P. Kabus, and R. Hartmann. *SeMoA Developer's Guide*. Fraunhofer Institute for Computer Graphics, Fraunhoferstraße 5, 64289 Darmstadt, Germany. Shipped with SeMoA software [10]. Locally availabe in the installation directory at docs/develop.

[25] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Co., 6th edition, August 2000. ISBN 020139815X.

[26] A. Spiegel. Object graph analysis. Technical Report B-99-11, Freie Universität Berlin, FB Mathematik und Informatik, Institut für Informatik, July 1999. Available at URL ftp://ftp, inf.fu-berlin.de/pub/reports/tr-b-99-11.ps.gz.

[27] A. Spiegel. Efficient distribution by static analysis. Unpublished. Available at URL http://www.inf.fu-berlin.de/~spiegel/eff.ps.gz, January 2000.

[28] Sun Microsystems, Inc. *Java™ Archive (JAR) Features*. in [30], relative URL: file:/docs/guide/jar/index.html.

[29] Sun Microsystems, Inc. *Java Language Specification*, 1998. Available at URL http://java.sun.com/docs/books/jls/html/index.html.

[30] Sun Microsystems, Inc. Java™ 2 SDK, Standard Edition, Version 1.4.1. Website, 2002. Available at URL http://java.sun.com/products/jdk1.3/.

[31] The OSGi Alliance. *OSGi Service Platform, Release 3*. IOS Press, Nieuwe Hemweg 6B, 1013 BG Amsterdam, The Netherlands, March 2003. ISBN 1-58603-311-5.

[32] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. Technical Report CoC 02-17, Center for Experimental Research in Computer Science, College of Computing, Georgia State University, Altanta, GA 30332, USA, 2002. Available at URL http://www.cc.gatech.edu/~yannis/j-orchestra/git-cc-02-17.pdf.

[33] University of Bologna. SOMA. Website. Available at URL http://www-lia.deis.unibo.it/Research/SOMA/.